VIRTUAL EXECUTION ENVIRONMENT FOR ROBUST REAL-TIME SYSTEMS

BY

KIHWAL LEE

B.S., Southern Illinois University at Carbondale, 1998

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Sceince
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Increasing demand for more features and bigger roles of software makes software more and more complex. Unfortunately, increase in the logical and organizational complexity of software generally leads to more bugs. Real-time embedded systems are not exceptions. Hastily written buggy firmware burned into a TV may not be easily "patched" by consumers. A best selling smart phone may be recalled due to the software bugs. These systems cannot be developed like safety-critical systems due to far more limited time and budget.

While many software engineering tools are useful in improving the quality of applications, the same tools are less efficient or inaccurate for system software. For controlling the residual software bugs in system software such as operating systems, we need more systematic enforcement in run-time. As a solution, we propose VEER, a Virtual Execution Environment for Robust real-time systems. VEER is based on a real-time virtual machine monitor (RT-VMM) and allows consolidation and partitioning of the subsystems of real-time embedded systems for better recoverability and service availability. Since RT-VMM has a complete control over the resource allocation, some of the important recovery can be made faster by appropriating resources for the process.

The recovery time is reduced through fault-containment, which is achieved by the following:

- RT-VMM based partitioning: The temporal and spatial partitioning

ensures certain execution faults are not propagated beyond the common execution boundaries.

- eSimplex, an analytic redundancy toolkit for embedded real-time systems: eSimplex enforces healthy component relations so that content errors are not propagated beyond component boundary, especially from less critical ones to more critical ones.

For restart recovery, process resurrection is used for fast and predictable recovery.

We examined the performance of VEER by comparing the existing real-time embedded systems and the migrated and reorganized version of the same systems. The performance overhead of RT-VMM is about 1.7 % if self-contained, or up to 4.7 % if networked in our experimental setup.

To my wife, Heejin.

Acknowledgments

I thank God for giving me hopes and strength to live. What good is an academic degree, if the burden and pain of your life suffocate you to death. Unfortunately it happened to some of my friends during my time in Urbana.

While excellence is valued and expected foremost in competitive environments, we must not overlook the people with difficulties who cannot express themselves due to the pressure of high expectations. We shall not allow ourselves to make excuses like "there were no warning signs" and leave indifference and pretense uncontested.

With his genuine passion, insight, caring heart and eternal optimism, my adviser, Professor Lui Sha, guided me through the long academic training at the University of Illinois. I would say, without any hesitation, having an academic adviser whom you can admire and trust has been the best part of my graduate studies.

I am also grateful to the other committee members, Prof. Geneva Belford, Prof. Marco Caccamo, and Prof. Yuanyuan Zhou. They all kindly and positively responded to my initial request and took time to help me to improve my work.

I would not be here without the support from my family and friends. My parents, Sangsup Lee and Jungmai Kim, never allowed their faith in me dwindle even when I myself was in doubt. Their prayer was what kept me going. My parents-in-law, Hongwoo Kim and Jaeryoen Do, also gave me love, support, and encouragement. My brother and sister-in-law, Kihwang Lee and

Youngsoo Jung, and brother-in-law, Dohyung Kim heartily supported me.

I would never forget the times I had with other Koreans in the department, especially the core members of *Mindalyun Society*, Dr. Sung-Il Pae, Dr. Hwangnam Kim, and Mr. Youngihn Koh. I also thank Dr. Won Jong Jeon and Dr. Sang-Chul Lee for valuable discussions on various research topics, which often lasted many hours.

My wife, Heejin Kim, and I went through difficult times together. It was her love, sacrifice and devotion that held our family together and got us through. Although she also pursued her own doctoral degree, her foremost concern was always the well-being of family. My daughter and son, Annette and Ben, always filled my heart with their love and smile. Dr. Heejin Kim's excellent ability to articulate and organize research issues was often of great help when her husband was overwhelmed and confused.

Strength and honour are her clothing; and she shall rejoice in time to come. She openeth her mouth with wisdom; and in her tongue is the law of kindness. She looketh well to the ways of her household, and eateth not the bread of idleness. Her children arise up, and call her blessed; her husband also, and he praiseth her. Many daughters have done virtuously, but thou excellest them all. Favour is deceitful, and beauty is vain: but a woman that feareth the Lord, she shall be praised. Give her of the fruit of her hands; and let her own works praise her in the gates. (Proverbs 31:25 – 31)

Table of Contents

List of Tables

List of Figures

1 Introduction

Computer reliability has always been an important issue since the early days of computing. For the first generation computers, the reliability of hardware was a major problem. Being made of thousands of electron tubes and environment-sensitive components such as mercury delay loops and Williams tubes, the first generation computers required engineers working around clock to keep the system operational. Despite the effort, the continuous up-time was relatively short when compared to today's computer systems until more modern technologies such as magnetic core memory were used to improve the hardware [7, 54].

Today's computer hardware is much more reliable but software has gotten very complex. It is becoming harder to build dependable software systems due to software's growing logical and organizational complexity. The growing complexity of software is due to increasing demands for features, and higher expectations on software capability and innovations. Software complexity makes it harder to reason about system-wide properties and thus introduces more chances of having residual bugs in production systems. It has been reported that debugging, testing, and verification processes range from 50 to 75 percent of the total development cost in a typical commercial development organization [35]. Real-time embedded systems are no exception. Many software projects ranging from cell phone software to complex avionics systems also face increasing difficulties in making the large integrated software work reliably [88, 71].

For this reason, a recent trend such as *Recovery Oriented Computing* [68] is gaining interests. For runtime software recovery, restart recovery techniques such as [20] are particularly considered effective because finding cures for the residual software bugs that cause failures is a costly operation. Wood reports that up to 80% of bugs that causes failures in production systems have no fix at the time of failure [89]. This signifies the importance of "post-mortem" recovery.

1.1 Practice of real-time embedded systems development

For safety-critical systems, it has been a common practice to separate software modules based on their *criticality levels*, so that the effect of less-critical component failures on critical components can be controlled. For example, the flight control system of an airplane, perhaps the most critical software in the whole system, is isolated from the cabin climate control system, a less critical one. This has been traditionally achieved by the Federated Architecture [76]. In this architecture, each functional unit is implemented in a separate piece of hardware, thus providing strong partitioning and fault isolation.

A more advanced architecture, Integrated Modular Avionics (IMA) [28, 76] runs all units on one computer or multiple computers connected by a common bus in a small box. The partitioning and isolation of this architecture allow software components with different criticality levels to run on the same machine or share more resources together. For example, Honeywell AIMS [3], the flight control system for Boeing 777, integrates many subsystems including flight management, communication, display, thrust management,

etc.

The presence of government standards and regulations such as FAA's DO-178B dictates how these kinds of systems are to be made dependable. The performance is often sacrificed in order to keep the software complexity under control so that the code can be analyzed, verified, and finally certified [38]. For government regulated safety-critical systems, it is required that the complexity of software (e.g. cyclomatic complexity [64]) is under a predefined threshold. This is to perfect the system in development phase, because any residual bugs in production systems can cause catastrophic events.

1.2 Dependability issues in real-time embedded systems

The fault-tolerance or recovery in real-time systems has primarily focused on hardware. Due to their stringent requirements on timing, roll-forward mechanisms have dominantly been used. For example, TMR (triple modular redundancy) is commonly used to detect and mask faults instead of rolling back. In predictable situations, small-scale software re-execution has also been used to overcome transient faults such as random bit-flip due to cosmic radiation. However, the fault detection is usually accomplished by running two identical copies of software in parallel and comparing the results [62]. Thus this type of fault detection is not meant to deal with software faults, which would occur on all of the copies at the same time in completely synchronized replicated systems.

In the real-time research community, researche has primarily been conducted on the scheduling aspect of real-time software fault-tolerance. Recovery scheduling issues were first systematically addressed and solved by

Imprecise Computation Model in [60], and recently this initial work was further developed and extended by Liverato et al. in [58]. However, Imprecise Computation Model's failure and task models are limited and do not capture real-life low-level issues such as process or OS crashes and recovery.

Apart from the scheduling-related works, it is hard to find established techniques or sustained efforts in real-time recovery. One of the early attempts to address the real-time software fault-tolerance issues at system-level is the framework developed by Anderson and Kight [5]. This framework provides error classification and tracking facility to invoke proper handlers. Most other run-time architectures are designed to convey the conditions and properties from the development time modeling and verification, which may insert various run-time checks. Even though these approaches are effective in enforcing predetermined properties, proving the correctness of actual binary code, not models, is hard. It can be made more effective with system-level run-time recovery mechanisms acting as safety nets to catch something that models might have missed.

Non-real-time fault-tolerance techniques have also been adopted to real-time systems. The concept of recovery blocks [74], the method that was traditionally regarded unsuitable for real-time systems, was also made possible to be used for real-time systems by distributed execution [45]. N-version programming (NVP) [6] is an attempt to use redundancy for software fault-masking, and it is a natural counterpart of TMR for software fault-tolerance. The most criticisms on NVP are based on the fact that even if multiple versions of software are developed separately, the faults may not be sufficiently independent. Analytic redundancy by Sha [80] also challenges the effectiveness of NVP by arguing that concentrated efforts on small core would yield a better result. While NVP requires one set of specification, analytic

redundancy requires separate specifications for different types of redundant modules. Simplex architecture [79] utilizes analytic redundancy and demonstrates its effectiveness for hard real-time systems.

1.3 Consolidated real-time environment

There are only few prior studies on integrated real-time environment, which is gaining interests after the adoption of IMA and perhaps by the influence of the renewed popularity of virtual machine monitors in server and desktop computing systems. SPIRIT-μkernel is one of the first consolidation architectures from academic research. Although it provides partitioning and allows multiple instances of operating systems to run on a single platform, it lacks full virtualization of resources including memory space. This results in ABI (Application Binary Interface) changes, but it does not affect compatibility much because its guest operating system only supports single address-space and binary reuse is not common anyway.

For scheduling in real-time virtual machines, Sha proposes TDM-like scheduling for the top level (VMM) and rate monotonic or cyclic executives for the low level (intra-VM) scheduling [81]. This work is the foundation of the operating environment that is approved by FAA [63] as a safe mechanism to migrate and consolidate existing real-time systems in virtual execution environments. Prior to this, consolidation of existing systems would mean complete recertification, which is extremely costly.

Real-time virtual execution environment has a great potential in improving software dependability by providing both temporal and spatial isolation between operating systems. Although there have been many efforts and great advancements using programming language techniques, formal methods, new

software engineering processes, etc., there is no cure-all tool. This is especially true in the case of system software. For instance, most of programming language-based techniques won't allow use of assembly language in the code, which is often inevitable in system software. Unless one builds an entirely new operating system such as Singularity [42], these tools are limited in its effectiveness when applied to the existing system software. In fact, even Singularity relies on the correctness of the basic core components, which are not fully machine-checkable. Constantly changing (e.g. new device drivers), complex, and highly optimized nature of system software also makes it harder to obtain efficient and accurate results using most tools that are otherwise very helpful in improving application software quality.

For low-level system software components, it is a growing consensus that we need a software architecture level of support for the isolation and recovery [85]. Real-time virtual machine architecture is strongly partitioned as suggested in [85] and shares many characteristics with microkernel systems.

For non-real-time systems, VMM has been used for fault-tolerance over a decade on modern computers. Bressoud and Schneider first implement the primary-backup approach on virtual machines instead of using multiple physical machines [13].

Due to recent interest in VMM, many studies have been conducted on VMM's role in fault-tolerance. Agbaria and Friedman in [2] used virtualized architecture to support checkpointing and migration among heterogeneous systems. LeVasseur et al. achieve device driver fault-containment by separating drivers in one or more virtual machines and executing using the original operating system for which they were written. This allows the use of unmodified drivers as well as improvement of dependability.

Because of the short history of real-time VMM (RT-VMM) and relatively

low volume of system-level recovery research activities in real-time academic community, no significant work has been done in this area. This thesis is one of the first attempts to leverage on the characteristics of RT-VMM in order to enhance the dependability of real-time embedded systems.

1.4 Overview of the solution

As an architectural solution, we propose VEER, a Virtual Execution Environment for Robust real-time systems. VEER is based on a real-time capable virtual machine monitor (RT-VMM) to enforce the isolation and virtualization at the hardware level. We have developed an RT-VMM as the core of VEER to show how recovery can be efficiently performed for real-time systems using virtual execution environments.

By allowing multiple real-time systems or subsystems to be safely consolidated, underutilized resources can be shared among virtual machines, and faster and more cost-effective processors can be used efficiently. Likewise, the "recovery slots" can be shared by multiple virtual machines and need not be duplicated for each one.

An ideal real-time virtual machine monitor (RT-VMM) would require no modification from existing real-time systems as long as it met their resource requirements. In the same sense, any existing software recovery mechanisms would also continue to work as effective as before in the new consolidated environment.

In order to minimize recovery times, and more importantly, to meet task deadlines, a careful planning of recovery is required. With the worst case of restart recovery times of tasks, we can change the CPU time allocation on failure. If we have enough slack, it might be possible to recover the failed

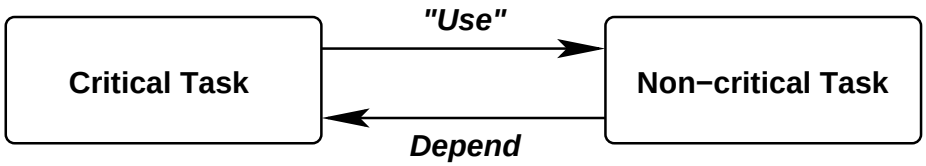

Figure 1.1: The Use relationship and Depend relationship. Critical components must not semantically depend on non-critical components

task and does not affect any other tasks. There are other cases in which the scheduler must dynamically adjust its scheduling policy. Instead of making schedulers complex, we utilize the strong and predictable temporal protection and partitioning capability of the real-time virtual machine architecture.

The most important rule to enforce in organizing components with varying degrees of criticality is that critical components shall not depend on the validity nor availability of service provided by non-critical components. From the view point of the traditional dependency notion, such a relationship still is a dependency at the syntactic level, but, at the semantic level, it is not a typical dependency. This is a weak form of dependency and we call it *Use* relationship [30].

Maintaining *Use* relationship has several benefits:

- **Functional independence.** Even if a critical component is syntactically dependent on a non-critical component, removal of non-critical component does not affect the correctness of the critical component.

- **Fault isolation.** Any type of failure including timing, value, crash, and hang does not propagate from a non-critical to a critical component. Although invalid data can flow between the two components, the critical component is not compromised by it.

- **Ability to maintain critical functions.** Critical components are

not affected by non-critical components, even if the failure rate of non-critical components is high. Therefore the system can still provide the critical core functions, leading to a higher availability. With the flat criticality model, this cannot be achieved.

As described above, maintaining *Use* relationship frees us from worrying about fault propagation. We can use an enhanced dependency graph based on failure semantics mappings [30] to determin the impact of a failure and pick the best candidate to restart. This process can be recursive as in [18].

In VEER, the overall system availability and service quality can be improved by utilizing the architectural characteristics and other basic design principles. The resulting systems can recover faster and some reorganization of subsystems can help mask the performance degradation during failure-recovery cycles. At a high level, VEER aims to achieve the following:

- Enable consolidation of multiple subsystems of existing or new real-time subsystems and systems.
- Enforce temporal and spacial partitioning for safety.
- Better utilize under-utilized resources through sharing.
- Maximize the service availability or quality through planned run-time recovery.

In summary, VEER allows better utilization of resource by consolidation and improves systems' robustness with dynamic recovery handling. Since the RT-VMM is fully virtualized, multiple operating systems can coexist in a machine. Compared to process-level or OS-level virtualization, this type of full virtualization can withstand OS-level failures. This is important because

the low-level system software is often difficult to analyze and verify due to its use of assembly code and highly optimized nature.

The rest of this thesis is organized as follows: Chapter 2 discusses the peculiarities of real-time recovery; Chapter 3 describes an efficient real-time restart mechanism; Chapter 4 describes how eSimplex implements analytic redundancy for embedded real-time systems; Chapter 5 deals with the principles and implementation of the real-time virtual machine monitor; Chapter 6 shows how RT-VMM is used for recovery; then we conclude in Chapter 7.

This thesis tries to adhere to the terminologies and definitions in [7].

2 Elements of real-time software recovery

2.1 Characteristics of real-time recovery

In order for real-time systems to be robust, all the error handling and recovery must be finished before deadlines. Although eventual recovery may be acceptable for non-real-time systems, it is in fact incorrect behavior for real-time systems. While this extra burden of timing exists, it also acts as clear execution boundaries within which the system can exercise more flexibility.

Consider availability of systems. For general purpose systems, a system with 99.999% uptime is considered as a highly available system. This number is translated roughly to 5 minutes of aggregated down time per year. Consistently limiting the downtime to under 5 minutes per year is not a trivial task. It involves a lot of testing, verification, and a heavy application of runtime fault-tolerance techniques in every part of the system.

Recovery Oriented Computing focuses on reducing MTTR in an attempt to more efficiently improve availability. In [19], Candea gives the following argument:

$$Availability = \frac{MTTF}{(MTTF + MTTR)}$$

and

$$\lim_{MTTR \to 0}(Availability) = 100\%$$

The availability can be improved by either increasing MTTF or decreasing MTTR. Obtainling a longer MTTF means a higher degree of perfection in development. Minimizing MTTR, on the other hand, is directly related to how recovery methods are designed and how the software is designed to accommodate the methods. Minimizing MTTR is especially appreciated because the availability can be improved even if we do not have any control over MTTF. This is often the case when COTS components are involved. However, no matter how hard we may try, obtaining MTTR of 0, thus reaching the availability of 1, is unlikely.

The situation is somewhat different for real-time systems. Real-time systems have an additional constraint, *deadline*. As long as deadlines are met and the results are acceptable, the system will be regarded as "working correctly," regardless of what is going on between the deadlines. If a real-time system is guaranteed to recover from any failure and produce acceptable results before deadlines, the availability will be 1. Suppose a task is running alone in a real-time environment and its worst execution time and the deadline (= period) is 1 and 3 seconds respectively. Suppose further that the task crashes near the end of its execution without generating output, and eventually produces a valid output after being immediately restarted, resulting 2 seconds of execution instead of 1 second. Even though the execution involves a crash and recovery, the task meets its deadline and produces a valid result, meeting all the real-time requirements. In terms of service availability, despite the failure, it has been available 100% of time. This unique property of

real-time systems allows us to plan for very high availability.

However, it does not mean that building reliable real-time systems is easier. In order to achieve the desired improvement for real-time systems, there are certain conditions to be met.

- *Fast and predictable detection and recovery mechanism.* As it is desirable to make everything predictable in real-time systems, recovery actions must also be predictable. Short execution times for such mechanisms are preferred, because shorter WCETs (Worst Case Execution Time) will lower the recovery overhead and make it possible to achieve higher processor utilization. Although detection mechanisms may not be able to detect and isolate all faults immediately, it will be acceptable if the symptoms or failures caused by faults are detected and corrected in time.

- *Reasonable coverage of detection methods* In order for a system to be robust, the detection coverage must be broad enough to catch most of common errors so that they do not propagate further. Some errors may be very hard to detect in time. In these cases, the method must translate the faults and their manifestation into another domain or detect them before the system behavior becomes unacceptable.

In the following sections, more detailed discussions on these issues will be presented.

2.1.1 Fast and predictable detection and recovery

Restart is a widely used technology for achieving high availability in general purpose computing. Advancements such as recursive restart [18] and fault avoidance techniques like software rejuvenation [40] have successfully

extended this technique for many more applications. For example, software rejuvenation techniques have been used not only for proprietary in-house software development, but also have been adopted for general-purpose systems such as IBM xSeries servers [21]. Restart technique has also been used in soft real-time systems such as telecommunication systems and manufacturing systems [16] where the temporary interruption of service is undesirable but acceptable.

Real-time embedded systems with hard deadlines pose a unique challenge in applying these recovery and prevention techniques. Their requirements are not limited to successfully detecting failures and restarting, but further demand the recovery to be efficient and predictable. For example, it is not reasonable to prescribe a simple shutdown and reboot remedy to the critical control systems. Real-time systems require real-time recovery.

Predictable and fast recovery mechanism is needed to allow the designers to reason about how the system will meet its timing requirements under failure-recovery scenarios. With the techniques like generalized rate monotonic analysis (GRMA) [78], the predictable overhead of recovery can be treated as a blocking time, thus one can make sure the system meets its real-time requirements even under recoverable failure conditions.

While predictability is one of the most distinguishing properties of any real-time recovery mechanism, devising such a mechanism is non-trivial due to difficulties in figuring the worst-case execution time (WCET) of the recovery procedure. Complex real-time systems typically consist of the real-time operating system, possibly some middleware and the application. It is not easy to acquire the worst-case execution time of the recovery activity. Most static methods will not be applicable in the presence of a complex modern processor and a 3rd party kernel with assembly code. What remains is

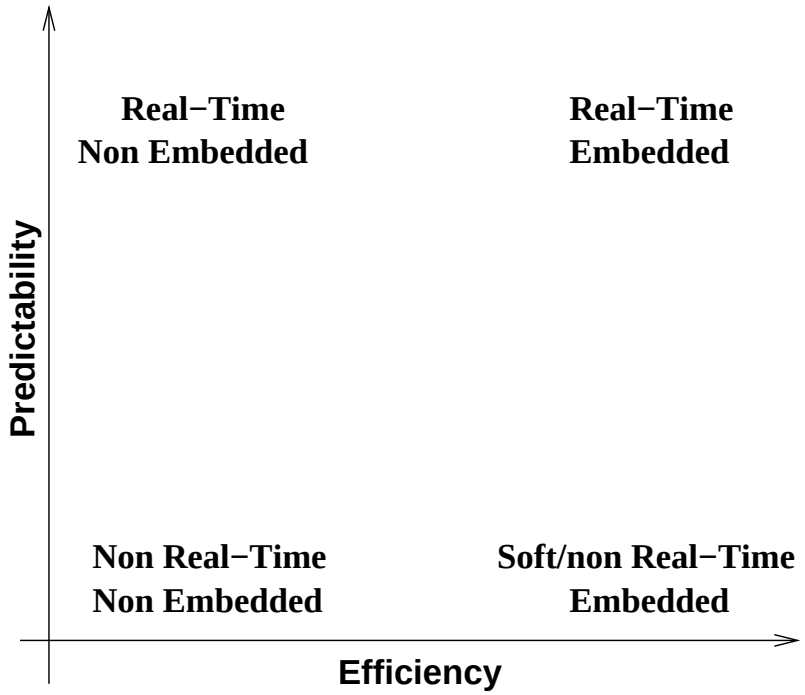

Figure 2.1: Dimensions of real-time recovery

measurement-based estimation, whose correctness is heavily dependent on the coverage of the test cases [47]. We will discuss how we can apply the recovery mechanism with good estimated WCET for hard real-time systems.

In addition to predictability, high efficiency is another desired feature for the embedded systems. Since such systems often leave not much room after performing normal duties, any recovery mechanism for small embedded systems with limited resources should not incur large processing or storage overhead. As a result, it is common to find a special customization of applications and commands in such embedded systems. However, since most software reliability and software fault tolerance techniques have targeted much larger scale applications and services, there are few or no tools that meet the need of real-time embedded systems.

Failure-restart mechanisms have long been implemented and used at the

hardware level. It has been commonly used even for non computer-related devices. For computers, the most common low-level form is the watchdog timer. For example, embedded microcontrollers like Atmel ATMega 128 have a built-in watchdog timer, which resets itself when the system fails to respond to the event properly. While this is effective in detecting system-wide crashes or hangs, extending its use to deal with specific applications is difficult.

The software heartbeat or periodic i-am-alive messages are the software version of hardware watchdog timers. It is possible to use a software heartbeat to detect a process crash and restart. In fact many restartable systems including [18] use this technique. When its overhead is acceptable, it is a simple but valuable tool for detecting crash failures and some hang conditions. However for high frequency hard real-time applications, the overhead can be too great.

QNX[37] and CHORUS/OS[1] support a restart of some subsystems and applications just as many microkernel operating systems do. QNX also provides hooks to recover the low-level details in the fault-tolerant version of system calls. Many other commercial embedded real-time operating systems also support traditional n+1 redundancy or process hot-swapping, primarily to work with hardware redundancy and fail-over.

Recursive restart [18] allows for a partial restart of the system with the help of a restart tree. It also tries to minimize the recovery time by taking the MTTF and MTTR values of components into account and dynamically organizing the restart tree.

In standard error handling in C/C++ programming, `setjmp()` and `longjmp()` have been widely used. `setjmp()` saves the state information, including the processor registers, at the specific execution point where it was invoked. Any subsequent `longjmp()` will restore that state information and the execution

will resume at the saved execution point. While it is fast and indeed effective for a certain class of failures, repeated use of this method can lead to software aging such as resource leaks. This is also not effective if the application terminates or hangs.

The concept of software restart has constantly been extended in the past decade by software rejuvenation [41, 21]. To avoid software aging-related errors, software rejuvenation resets a running process before the occurrence of any failure. A considerable amount of research has been conducted on modeling of the software system and its aging to efficiently determine when to perform the rejuvenation [86].

Libft and watchd [40] offer monitoring and a reusable recovery facility for restartable systems. For soft real-time systems with less resource constraints, the latency of common monitoring may be acceptable.

Although there have been great number of researches conducted in restart recovery, it is very rare to find anything directly related to real-time systems. In the real-time research community, investigation has been done primarily on the scheduling aspect of real-time software fault tolerance The recovery scheduling issues were first systematically addressed and solved by Imprecise Computation Model in [60], and recently it was further developed and extended by Liverato et al. in [58]. There are other important works, but we will not delve into this subject as we are focusing on the system-level issues, not scheduling issues.

The performance and the predictability are the most important factors to consider in designing real-time systems. Recovery mechanisms, being part of real-time systems, also need to be predictable and exhibit reasonable performance. As a building block of robust real-time systems, we have developed *Process Resurrection*, a fast real-time recovery mechanism. The details will

be presented in Chapter 3.

2.1.2 Coverage of detection methods

Robust systems must have failure/error detection mechanisms with comprehensive coverage. For each and every possible failure mode in the system, there must be corresponding detection mechanisms capable of spotting erroneous system conditions before system becomes unrecoverable. For real-time systems, any unacceptable delay or fail-stop (hang, crash, etc.) can be translated to a deadline miss, a special case of timing errors. Given that resources are partitioned and protected, this aggregates many failure modes and simplifies the detection process. But not all systems can afford to miss a deadline then recover.

For real-time hardware fault-tolerance, fault masking techniques such as TMR (triple modular redundancy) has been used to prevent any deadline miss. For software, we will also use a form of redundancy called analytic redundancy [80]. In analytic redundancy, a small well-tested core component implements the base minimum essential functions the system must provide. One or more redundant components can provide more complex features that are not fully tested or validated. By using carefully designed acceptance test, the healthiness of the outputs from feature-rich components can be assessed. It can be a simple range check with regard to the corresponding output from the core component, or it can be as elaborate as LMI-based [12] safety tests for control systems. The detection is immediate and faults are masked by using the results from core components. Note that use of the results from core components likely means degraded performance, since they lack features or refinements for the sake of simplicity.

What about the coverage? The use of analytic redundancy is not just

for fast detection and processing of timing failures. Since it tests the acceptability of outputs, content errors are also covered. Furthermore, it translates timing errors to content errors. Absence or unupdated outputs are tested against the results from core components, and the errors will be detected if the test shows the results are unsafe. Given that the simple core component is well tested and highly reliable and protected temporally and spatially from potentially faulty components, analytic redundancy can take care of just about any software errors.

VEER utilizes analytic redundancy in order to maximize the coverage of failure detection on critical real-time components. We have designed an efficient analytic redundancy framework for embedded systems. The details are shown in Chapter 4.

2.2 Fault containment and protection

Detection of and recovery from faults and failures are meaningless if faults are propagated through unexpected paths to other components. While making components work correctly is the most important task in achieving a high dependability, designing components less susceptible to faults also help control erroneous conditions. Hence having robust system components helps enhancing fault containment.

2.2.1 Robustness in real-time software

Robustness is defined as "dependability with respect to external faults" [7]. Robust systems are resilient and resistant to unexpected or incorrect external events. When we apply this concept to smaller entities that makes up a system, we call it *modular robustness*.

Modular robustness protects components from contaminated by external faults including incorrect inputs and prevents further propagation of faults (i.e. fault-containment). On the contrary, fragile systems or components lack extensive and active handling of unexpected events, so they become reliant on non-persistent conditions such as network connections, remote service availability, etc. It is very easy to end up with a fragile system if the designer assumes everything will work as planned. Robust systems are capable of withstanding or handling unexpected events and continue to function at an acceptable level of performance. Additionally, robust real-time embedded software must deal with the unexpected conditions in timely manner with limited resources.

One of the key concept in preventing fault-contamination is *Use* and *Depend*. While any communication of data between two components results in a dependency in the traditional sense, it is syntactic level dependency. In semantic level, there may be no real dependency in terms of functional reliance for acceptable execution. In this case we call it *Use* relationship. If a component is actually functionally relying on the other component for producing acceptable outputs, we call is *Depend* relationship. This concept has been used informally for many years, but recently there was an attempt to formally define these using failure semantics mappings [29].

2.2.2 Protection and Partitioning in Recovery

The level of protection between runtime units varies from one architecture to another. The ideal architecture for robust real-time systems must provide i) strong isolation in both temporal and spacial dimensions with a low overhead, and ii) ways to control the allocation of time and space in predictable manner.

Software components must be protected from each other in terms of time

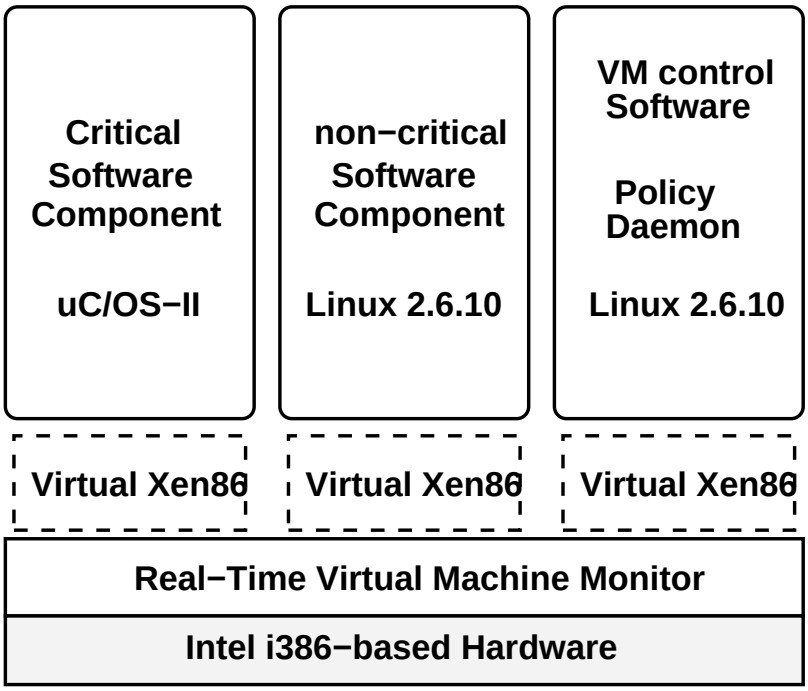

Figure 2.2: An example of virtual machine architecture

and space. Even if one task overruns or falls into an infinite loop, other tasks must be able to use the promised share of processor time. This property is maintained by real-time task schedulers. Similarly, when a task exceeds an array boundary and tries to overwrite a memory area that belongs to another task, it should be detected immediately and not be allowed. This is commonly done by a virtual memory manager with a hardware memory management unit (MMU). These are what most of modern operating systems offer, but the degree of isolation provided by these operating systems is not sufficient. For example, if a task is preempted during a system call, the OS state presented to the next task can be inconsistent and is difficult to prove that the inconsistent portion of the OS state space does not affect other tasks.

For stronger isolation and protection, use of a virtual machine architecture is desired [44]. Since it is meant to provide each of multiple software systems with an illusion of being the sole user of its own physical machine,

the greatest partitioning is also provided. The strong partitioning between healthy and failing tasks, and also between critical and non-critical tasks will make the interactions between tasks explicit and clearly visible. This is a very important property for enforcing *Use* relationship between tasks with varying degrees of criticality.

If a virtual machine running on top of a real-time virtual machine monitor is guaranteed to get a predefined amount of CPU time in a specific time interval, CPU usage changes outside the virtual machine activation period will not affect it in any way. This is achieved by the combination of temporal and spacial partitioning and isolation. When a failure of a critical task occurs, we can choose from several different recovery mechanisms and possibly assign more time slots for faster recovery to meet the end-to-end deadline. One extreme scenario is that when a critical component needs to be recovered urgently and the CPU time "reserve" is not enough, we can pick one of non-critical component and take allocated CPU time away to assign it to the recovery activity. By employing a synchronous and configurable top-level scheduler for virtual machine scheduling, we can safely achieve this without affecting any other tasks.

3 Fast Restart Recovery

In this chapter, we demonstrate how one can build an efficient and predictable recovery mechanism for real-time embedded systems with limited resources. We have developed *Process Resurrection*, a restart mechanism, with which an application can be made to automatically restart on crash failures. The restart can also be initiated on demand to provide software rejuvenation or upgrades. We applied this technique to the inverted pendulum control system and an mp3 audio playback software. We also show how to incorporate the measurement-based WCET estimation of *Process Resurrection* to hard real-time systems

3.1 Architecture

For efficiency and performance, the recovery mechanism should introduce only a small amount of additional code to the system. It translates to a low

API	Description
`r_initialize()`	Set up a handler for the specified signals and register user defined pre-processor
`r_malloc()`	Allocate the specified size of state storage
`r_free()`	Deallocate the state storage
`r_pthread_mutex_lock()`	BCP-enabled lock (optional)
`r_pthread_mutex_trylock()`	BCP-enabled trylock (optional)
`r_pthread_mutex_unlock()`	BCP-enabled unlock (optional)

Table 3.1: Process Resurrection Functions

Signal	Description
SIGFPE	Erroneous arithmetic operation.
SIGILL	Illegal instruction.
SIGSEGV	Invalid memory reference.
SIGBUS	Access to an undefined portion of a memory object.
SIGSYS	Bad system call.
SIGXCPU	CPU time limit exceeded.
SIGXFSZ	File size limit exceeded.

Table 3.2: POSIX Signals used for notification of impending crash.

run-time overhead and storage requirement. We have achieved this goal by utilizing features already existing in the operating system and the processor. Table 3.1 lists the functions provided by *Process Resurrection*.

3.1.1 Crash Detection in Operating System

The crash detection in *Process Resurrection* is directly linked to the crash detection mechanism of the processor and the operating system. Unix-like operating systems offer signals as an event notification mechanism. They are used for normal events like job control (SIGINT, SIGTSTP, SIGKILL, SIGCONT, etc.) as well as for the notification of an abnormal state detected by the hardware or the operating system. Among those error notification signals, the most common one is segmentation fault (SIGSEGV). Others in this class include bus error (SIGBUS) from address misalignment, arithmetic operation error (SIGFPE) like divide by zero, and execution of an illegal instruction(SIGILL). Table 3.2 lists the signals used for erroneous event notification.

These signals notify the offending process of fatal errors that will make it no longer operational. Even if the signal is handled, the behavior after re-

turning from the signal handler is undefined in POSIX. Nevertheless, on most Unix-like operating systems, the process will be terminated after returning from the handler. This form of termination is commonly called application crash.

Although one can assign a custom signal handler for most of the signals, the crash signals are usually not directly linked to a recovery action. Instead of using special user-level handlers, it is common practice to simply use the default signal handler for such signals, which cleans itself up, dumps core if possible and exits. Some applications try to display user-friendly messages or launch quality feedback software.

Instead of terminating, *Process Resurrection* uses these crash-related signals to initiate the application recovery. They have not only an excellent coverage of process crash failures, but also a very short delay between fault occurrence and its notification: a quality favorable to the recovery of real-time embedded systems.

The operating system also notifies applications of non-fatal errors in the form of the return value of system calls. These values and exceptions must be checked and handled by the application. Without comprehensive handling of such conditions, the application may exit unexpectedly or eventually cause crash failures that may happen repeatedly even after restarting.

3.1.2 Triggering the Recovery

The common way of automating restart is to have an external monitor periodically check the status of the process and launch the process again if a crash is detected. While this technique is widely used for non-real-time applications, its run-time overhead and fault detection latency is too much for real-time embedded systems. To reduce the fault detection latency for a

control system, one may have to check the process for every control loop. If not checked frequently enough, an overlooked crash will delay the recovery and miss the deadline. This is the reason why fault notification signals are used in our system instead of monitoring.

The signals are very efficient recovery triggers; in Linux there is no normal case run-time overhead and the operating system activates the signal handler, through `do_signal()` kernel function, only when a failure occurs.

In *Process Resurrection*, we assign a special signal handler for every crash-related signal to trigger the recovery. The application developer may choose to ignore or assign handlers to some of the non-fatal signals whose default action is to terminate the process. If some of these signals are used for normal execution, the signal mask must be set up correctly in order for the process to handle the normal signals, even after a recovery.

A recovery is triggered when one of the crash signals is delivered to the process. Once the operating system sends the signal to the process, the registered handler will be executed instead of returning to the preempted point. Inside the custom signal handler, the following tasks are performed:

- Do any user-defined pre-processing.
- Gather the state information to transfer, if necessary.
- Replace (`execve()`) the process image with its own or an alternative and transfer the state information.

When the process image of the offending process is replaced in the last step, all the locally held resources of the offending process will be automatically released by the operating system. This is a desired property for dealing with software aging-related failures. The shared resources, on the other hand,

may require special handling, as other processes may expect the resources to be in a certain state. Since the image of the offending process is considered corrupt, any complex clean-up action on the shared resources may lead to another critical error. For this reason, such actions are taken after the replacement.

Application-specific recovery action is selected based on the information transferred when replacing the process image. Such information is propagated in the form of arguments when invoking the `execve()` system call. For example, when the control process crashes in our inverted pendulum control system, we transfer the information, such as sensor calibration data, in this way. The techniques for transferring more complex information are explained in the next section.

The replaced process image need not be of the same process. One can choose to replace the current process image with an alternative image. It is useful when a process repeatedly fails, one resurrection after another. The user can set a limit on the number of consecutive resurrections and specify an alternative version that will be used when the recovery reaches the limit. This feature can also be used for non-failure conditions such as reconfiguration or upgrade.

The process replacement of this form is very efficient. The restart is immediate and the binary loading time is often significantly reduced when the system is equipped with a fast and large cache. Since the process retains its process id, unlike the terminate-and-restart method, some of the restart dependencies between processes can also be reduced.

In our system, the `r_initialize()` function allows users to specify which signals to be associated with the *Process Resurrection* signal handler. Users can also define custom pre-processing and post-processing for the recovery.

However, some system functions, including mutex operations, are not async-signal safe, meaning that use of such functions in a signal handler may cause unexpected results, including deadlock. POSIX guarantees certain functions to be async-signal-safe. The user-defined pre-processing routine must use only such system calls as listed in Table 3.3.

3.1.3 Persistent Storage and State Transfer

Process Resurrection can be used with number of existing checkpointing mechanisms. However, the conventional way of accessing persistent storage (i.e. a disk) may not be desirable to real-time embedded systems due to its slow speed and long blocking effect. To overcome this, memory-based persistent storage is provided for storing local state information. The lifetime can be as long as that of the operating system. Since we are interested in process-level recovery, such a degree of persistency is sufficient.

The memory-based persistent storage was implemented using shared memory. On Linux, shared memory is allocated from the kernel memory space and then is mapped to the virtual address space of a user space process. Once mapped, the access involves no system call or context switch. Furthermore, it will not be paged out, as the paging on the allocated region will be disabled when created. The shared memory is preserved even after the creator process has crashed. Unless the system call that destroys the shared memory is explicitly issued, it will remain until the operating system reboots.

The resurrected processes can remap the shared memory region to its address space after recovery, at a very low cost. The way the state information is retrieved can be dictated by the combination of the argument given at the time of image replacement and the predefined protocol by the user.

The persistent storage can be allocated by calling the `r_malloc()` func-

_exit()	fstat()	read()
sysconf()	access()	getegid()
rename()	tcdrain()	alarm()
geteuid()	rmdir()	tcflow()
cfgetispeed()	getgid()	setgid()
tcflush()	cfgetospeed()	getgroups()
setpgid()	tcgetattr()	cfsetispeed()
getpgrp()	setsid()	tcgetpgrp()
cfsetospeed()	getpid()	setuid()
tcsendbreak()	chdir()	getppid()
sigaction()	tcsetattr()	chmod()
getuid()	sigaddset()	tcsetpgrp()
chown()	kill()	sigdelset()
time()	close()	link()
sigemptyset()	times()	creat()
lseek()	sigfillset()	umask()
dup2()	mkdir()	sigismember()
uname()	dup()	mkfifo()
sigpending()	unlink()	execle()
open()	sigprocmask()	utime()
execve()	pathconf()	sigsuspend()
wait()	fcntl()	pause()
sleep()	waitpid()	fork()
pipe()	stat()	write()

Table 3.3: Async-Signal Safe Functions

tion that returns the pointer to the storage of requested size. It is rarely necessary but the allocated persistent storage can be deallocated using the `r_free()` function. Once allocated, the storage can be used in the same way the local variables are used, but it is recommended that users define a structure to organize the storage.

Our recovery mechanism has no support for checkpointing [40] to save and restore the complete application state. This is because our target application domain is primarily real-time applications that require simple state restoration. Some applications may have soft states, which the process can reacquire or approximate after the recovery. For the inverted pendulum control, for example, the pendulum's angle and track position are reacquired at every period, so losing the current input is of no concern. The essential state information includes the device-specific calibration data that can be obtained only at the initial state of the pendulum. The filtering history is an example of a soft state. Even if the filtering history is lost, the filter will again be stabilized after a number of control loop executions. Another reason why we do not provide a generic checkpointing facility is that it is less effective than application-specific methods [22].

However, some state information cannot be transferred by using only the application level mechanisms. For example, TCP connection information will be stored in the kernel, and it will be lost after the process is reinitialized. To reconstruct the communication channel with no changes in the kernel, an application-specific user-level recovery protocol must be used. The kernel-level monitoring and logging such as FT-TCP [4] can also be used in conjunction with *Process Resurrection* to provide a transparent communication channel recovery. The same design principle applies to the non-persistent IPC mechanisms.

During the resurrection, all normal activities including communication, computation, and normal signal handling will be suppressed. If the communication peer tries to communicate with the process being resurrected, the message will be either blocked or discarded. After the completion of the resurrection, the communication channel may be destroyed if it is a non-persistent one. Thus the application should be able to anticipate such an outage and to reconstruct the channel.

For easier recovery, use of an asynchronous state-less protocol is preferred. However, low-level communication semantics should not be more restrictive than application level semantics, in order to minimize extra effort for recovery. For example, if the high-level application semantics allows a message to be lost, the low-level communication can be UDP-based without any explicit recovery mechanism. Using TCP for such an application will involve additional exception handling code to make it restartable. For applications with synchronous communication semantics, a synchronous low-level communication mechanism like TCP will be more suitable. To make it restartable, however, additional application-level exception handling and recovery are required. One can, of course, use a transparent recovery solution like FT-TCP.

3.1.4 Preventing Unbounded Priority Inversion

In priority scheduling-based systems, multiple tasks that share resources may exhibit unbounded priority inversion, a lengthy blocking of a higher priority task by a lower priority one. To help make the system more predictable, a user-level call wrapper for locking and unlocking is also provided. Inside the wrapper, a priority inversion [77] avoidance technique similar to the priority ceiling protocol is implemented. If the underlying operating system supports a priority inversion avoidance protocol, the native locking and unlocking calls

can be used directly.

The protocol raises the priority of calling process to the predefined resource priority that is higher than that of any normal process. When there are only those tasks that belong to one resource-sharing group, the maximum blocking time will be identical to the priority ceiling protocol. However, under more complex scenarios, a high priority task may be blocked by the duration of an unrelated critical section. However, users are allowed to override the predefined fixed resource priority in order to implement the ceiling protocol.

3.1.5 Predictability of Process Resurrection

Since most of *Process Resurrection* consists of system calls, the timing analysis has to deal with the kernel code execution, which is not easily achievable with static analysis methods. The difficulty of obtaining a precise WCET using static methods lies in the very nature of kernel code; it includes assembly code, it is complex, and optimization is heavily used. As a result, we settled for a conservative WCET value and extend the deadline using Kalman filter if the actual WCET exceeds the estimate. This technique has been demonstrated in more complex control systems such as [34].

In the next section, we will show a control system with a tight budget as an application.

3.2 Experiments

We have implemented an inverted pendulum control system (Figure 3.1) and an MP3 audio player that can recover from crash failures using the *Process Resurrection* technique. For demonstration purposes, they only deal

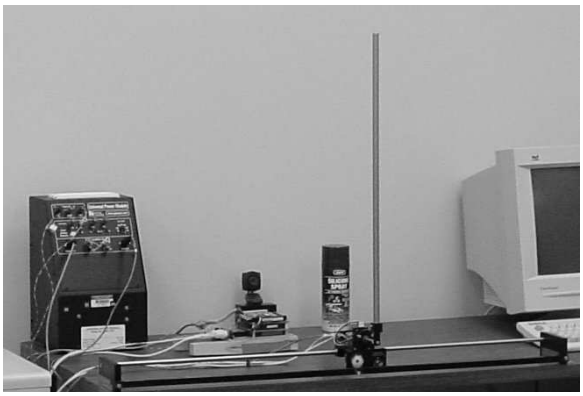

Figure 3.1: The inverted pendulum control system

with segmentation faults, but they can be made to recover from other crash failures by registering the trigger handler for other types of crash-related signals. In addition to a crash failure recovery scenario, *Process Resurrection* is applied to a software rejuvenation of the Telelab system, a remote control laboratory that allows on-line upgrades of the control software.

3.2.1 Inverted Pendulum Control

The inverted pendulum control system is used to demonstrate the agility of *Process Resurrection*. This control system is inherently instable. Even when the angle error and angular velocity are small, several missed control outputs can make the pendulum fall. When the pendulum is tilted past a certain angle, an even one missing control outputs can lead the pendulum to crash. Due to this strict timing requirement, the detection of control software's crash and recovery from the crash must be carried out in a timely and predictable manner.

The control computer is based on AMD's 133 MHz Élan SC520 processor, which is an 80486-grade processor with 66 MHz memory bus. It has 16 KB of level 1 cache on chip, but no level 2 cache is present on the board. The

computer is equipped with the quadrature encoder interface for sensing input and a digital to analog converter for control output. It runs the Linux kernel with RT scheduling and kernel preemption enhancements. While the kernel is being loaded, the compressed root file system image in the flash memory is decompressed to the ram disk. So the file system accesses following the boot process are indeed main memory accesses.

Single Control Process

In its normal operation mode, the control process acquires sensor inputs for the angle and the track position, and then calculates the control command that is subsequently sent to the D/A converter. The control frequency is 50 Hz. In other words, these operations are repeatedly performed every 20 ms. It means that the fault detection, notification, recovery, and re-execution of the control software must be finished in less than 20 ms, in order to avoid missing any control output.

To meet this timing requirement, *Process Resurrection* is used. When a SIGSEGV signal is delivered and the process is about to crash, the recovery handler preempts normal execution of the control software. The handler will then initiate the replacement of the process image using an `execve()` system call. As an argument to the system call, the device calibration data is transferred in order to prevent the resurrected process from erratically calibrating the device again at the non-initial plant state. In most cases, this is enough to maintain control, but a more seamless recovery requires additional data, such as the history concerning the input-filtering algorithm. Since this information consists of a number of floating point variables, it is written to the already created persistent storage with `r_malloc()`. Once the operating system initializes the resurrected process, the process will examine

	Case 1	Case 2
Avg. fault interval	510 ms	20 ms
Range	20 - 1000 ms	10 ms - 30 ms
With deadline extension	no effect	little jitter
With full dependency	very jittery	unstable

Table 3.4: Results of simple fault injection. Each case ran for 1 hour. The unstable case didn't complete the 1-hour run.

the arguments to `main()` and recover the state information stored in the persistent storage.

In order to see how the recovery affects the schedulability, the execution times were measured using an additional 8254-based programmable interval timer on the I/O board. The resolution of the timer is 125 ns. As shown in Table 3.5, the time it takes to notify the process of the impending crash (i.e. signal delivery) is 38 μs in the worst case. The time spent on replacing the process image and loading the state information is 11.2 ms in the worst case. The standard deviation is smaller than the one measured on the Pentium III 1.2 GHz machine, mainly because it lacks the level 2 cache. Since the control itself takes 25.75 μs, the entire recovery and normal execution time is under 12 ms, which is far less than our 20 ms of budget. In this case, even a very conservative WCET estimation can be accommodated. For details see Table 3.5.

Limited Budget

The situation, however, changes when we have more than one process and a tighter loop. In eSimplex [59] systems, multiple versions of controllers are running. The semantic error monitor with a safety controller ensures the stability of the plant, even if an untrusted controller is uploaded as an upgrade. Although the user controller is still running with 20 ms period, the

deadline is 10 ms, because the control commands from trusted and untrusted controllers must be checked for safety by the eSimplex core task, a higher priority job, before outputting the chosen command to the actuator by the end of the 20 ms period.

From the previous measurements, we estimated the WCET to be 12ms, giving about 6 % of over-estimation. With this WCET, the crash detection and recovery will not fit in the budget of 10 ms. When the user controller crashes, it will miss the deadline after recovery. Although the system can maintain stability in the presence of an occasional deadline miss, frequent misses will surely bring the system down. In TableIV, the effect of frequent misses is shown. The test was conducted with two different failure rates and each case ran for one hour with the exception of one case, which crashed within seconds. [1] When the crash is uniformly distributed between 10 and 30 ms, the control will fail to maintain the stability of the inverted pendulum. Even if the failure rate is significantly smaller, the control performance will be noticeably degraded.

The control failure or performance degradation is due to the tight deadline in the system. When the control process misses deadline, it is preempted by the higher priority task, breaking the timing dependency between the controller and the sensor. To solve this problem, a Kalman filter is used to extend the deadline. The Kalman filter will smooth out any sudden changes in the input values that are unusual in continuous physical systems. The improved performance is shown in Table 3.4. Under the same stress conditions, the system with a weakened dependency performs significantly better. Even in a high failure rate scenario, it maintains stability instead of failing. When

[1] The video clip of the experiment are available at http://www-rtsl.cs.uiuc.edu/pr/

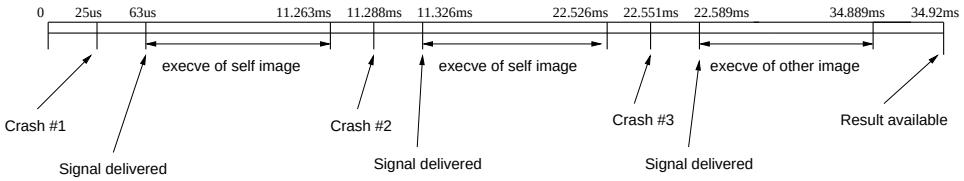

Figure 3.2: Using alternative process image

the failure rate is lower, the crash failures exhibit no perceivable effect on the quality of the control.

The deadline extension involves a state transfer to the resurrected process. The filter data structure is stored in the persistent storage and the resurrected process continues to use the data for filtering. The addition of persistent storage incurs one page (4 KB) of additional memory overhead and a small initial setup time, which is in the order of microseconds in our configuration. No normal case run-time overhead is added. When the estimated WCET based on measurements is close to the time budget, we can apply the same deadline extension techniques to minimize the effect of budget overrun.

Using an Alternative

Process Resurrection can also replace the process image with an alternative. In an inverted control system, the pendulum may be stable enough to be controlled even after one missed period. If we are allowed to miss one deadline, we can afford to have more than one crash failure in a period. It is useful when the controller repeatedly crashes in the current system state. See Figure 3.2. Suppose the control process started at time 0 and crashes at time 25 μs, at nearly the end of its execution. The process will then be resurrected at 25 μs + 38 μs + 11.2 ms = 11.263 ms. However, it crashes again after 25 μs just like the first time. If we resurrect the process again, the next execution will complete at around 22.551 ms. Since we are allowed to miss

at most only one deadline, it will not be a problem. However, if the freshly resurrected process fails again for the third time, it is very probable that the subsequently resurrected process will also fail. For this reason, we allow setting the limit on the number of resurrections per user defined period. The limit is set to two per 30 ms in this example. Suppose, at time 22.551 ms, the second resurrected process crashes. Since we have set the limit to two, it will be replaced with another version, which is the last known working version. As shown in Table 3.5, replacing with a different process image takes a little longer, 12.3 ms. As a result, the third resurrection will be completed at 34.889 ms. Assuming the execution time of the alternative version is 31 μs, which is not significantly longer than the original version, the computation will end by 34.92 ms, well before the second deadline, 40 ms. Accordingly, the version switching prevented a possibly endless resurrection cycle due to a critical bug in the original control software.

3.2.2 MP3 Audio Player

We have also applied the *Process Resurrection* technique to an MP3 audio player. In fact, the original mp3 player we modified is a sufficiently mature application. Thus, we artificially injected faults to demonstrate how the technique would work with such an application. Although such an application is considered as soft real-time at best, reducing perceivable disruption is also desirable in the recovery.

The audio player chosen is *mpg123* version 0.59r. We have modified the source code to apply our technique. As in the previous example, a segmentation fault will trigger the process image to be reinitialized. Between the resurrections, the name of the mp3 file being played and the location inside the file at the time of crash are transferred.

Even with no real-time support of the operating system and no optimization for reducing the initialization time after recovery, the perceived sound quality degradation was small. [2] If the application is further modified to reduce the recovery time, it may be possible to have no loss in the sound quality. However, for simplicity we simply added a recovery handler and a simple state transfer using the arguments to the system call.

When the segmentation fault was injected roughly every 5 seconds, the player recovered and continued to play the music. Although we have not applied any dependency weakening technique, the effect of disruption is reduced by the buffer in the sound card, which can hold several hundred milliseconds worth of sound data at the sampling rate of 44.1 KHz. The buffering is also a known timing dependency weakening measure. Since the buffer resides on separate hardware and is not destroyed by the process restart, it has property of timed persistency. Thus if we recover and feed the buffer before it runs out, there will be no perceivable disruption in the sound quality.

3.2.3 Software Rejuvenation

The Telelab [59] is a remote control experiment system that allows users to upgrade the control software on the fly and to monitor the effect of the upgrade. The underlying Simplex architecture [80] detects semantics, timing, and execution errors of the control system before the defects of the new software break the control.

Although the control is never compromised thanks to the Simplex architecture, the additional services that interface the system with a user may show signs of aging after many upgrade requests are carried out. In the be-

[2] The recording of the experiment is available at http://www-rtsl.cs.uiuc.edu/pr/

Action	AMD SC520 133MHz (486)	Intel Pentium III 1.2GHz
control calculation	24.51μs (0.54)	3.03μs (0.35)
	25.75μs	3.74μs
execve()-self	10.7ms (0.11)	601μs (19.98)
	11.2ms	680μs
execve()-other	11.7ms (0.18)	980μs (22.55)
	12.3ms	1078μs
fork()	409μs	158μs
	454μs	166μs
signal	37.15μs (0.35)	6.0μs (1.19)
delivery	38.0μs	8.12μs

Table 3.5: Measured execution times. Average (std deviation) and worst case are shown.

ginning, a periodic rebooting of the system is in place, but it would become unavailable if an unusually high number of requests were made during the reset period. This actually occurred when a large number of people accessed the system as a class project near the end of semester. Rather than fixing the complex distributed software, a software rejuvenation technique was used with the help of *Process Resurrection*.

In the new system, an external monitor keeps track of the number of upgrade requests made and of clients connected. When the numbers reach a predefined threshold, a signal is sent to some of the processes constituting the system in a certain order. Each process in the system is modified using our technique, so that they can be quickly restarted upon reception of the signal from the monitor. Since the system is now refreshed based on a more relevant aging factor and the restart is fast, the availability of the experiment system has been greatly improved.

CPU	Speed (MHz)	Cache	Repl.
Pentium III	1200	512K L2	1.07 ms
Pentium III	1000	256K L2	0.82ms
Pentium III	733	256K L2	1.60 ms
AMD K-6 II	533	256K L2	1.77 ms
UltraSparc Iii	360	256K L2	4.50 ms
AMD SC520	133	n/a	12.3 ms

Table 3.6: Image replacement times

3.3 Performance

The values shown in Table 3.5 were measured under the typical system load. For the embedded computer, the control software was running at a higher priority level and occasional UDP-based traffic was present. The measurement was performed using the `RDTSC` instruction on Pentiums and AMD K-6 II. For UltraSparc, the `gethrtimer()` function of Solaris was used. Finally for AMD SC520, which lacks native timing measurement support, an additional hardware with an 8254 timer was used. The computer with UltraSparc was running Sun Solaris 7 and others were running GNU/Linux.

3.3.1 Replacement Time

The fault-detection to signal delivery time was relatively short when compared to the time to finish `execve()`. Since `execve()` involves accessing the file, it will take much longer than a straightforward execution of the kernel code. With a fast and large L2 cache on the Pentium III Tualatin processor, the image reloading time (680 μs) is much shorter than the time to load other uncached images (1078 μs). The difference is much smaller on the embedded computer (10.7 ms Vs. 11.2 ms) that has no level 2 cache.

The measured values for using `execve()` to replace its own with an alternative image are shown in TableVI. It shows that when loading an uncached

image, the processor speed is not the dominant factor in timing. Since the replacement involves uncached data access from the file system, the relatively long disk access speed will determine how soon it finishes. In fact the 1200 Mhz machine was slower than a 1000 Mhz machine with an even larger cache. It is because the 1200 Mhz machine has a slower disk subsystem (hdparm -t value of 20.98 MB/s) while the 1000 Mhz system (hdparm -t value of 52.03 MB/s) is equipped with Ultra 160 SCSI controllers and disks. AMD SC520, the control computer in our inverted pendulum experiment, avoids a possibly much longer execution time by keeping the file system in the ram disk.

3.3.2 Storage Requirment

The use of *Process Resurrection* with simple or no state transfer adds very little extra code and data. The binary size of the original inverted pendulum control is 14,871 bytes unstripped, and the resurrection-enabled version is 15,148 bytes, adding only 277 bytes. When the binaries are stripped, the difference is only 20 bytes.

The control computer's persistent storage is based on a small flash memory, where the kernel and root image are stored. Adding 20 bytes to the root image is not considered as an overhead in almost all cases.

3.3.3 Processor Overhead

The extra execution time of the recovery mechanism is added only when there is a crash. Assuming the time between crashes is reasonably long for well-tested systems, the overall run-time overhead is very small. If periodic monitoring is to be used instead, run-time overhead will be present even when there is no failure. In our system, most of the remaining CPU time

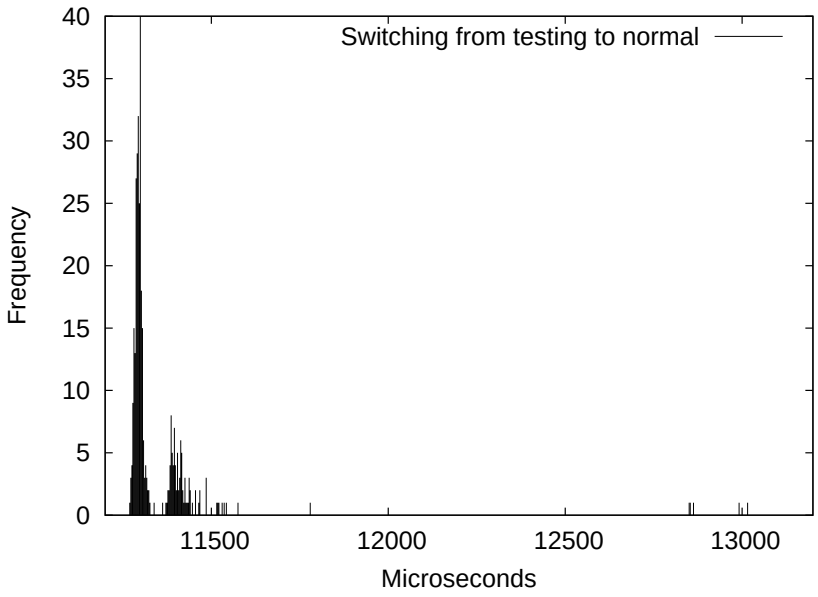

Figure 3.3: Replacement time when using alternative process image

can be used for other non real-time jobs or soft real-time applications.

For different types of applications, the actual recovery time will vary because the post-processing after the process image replacement is application-dependent. If the application is stateless or only has soft-states, the post-processing will be minimal. If the application requires large dynamic resources, the initialization may take a long time, adding even more time for recovery to become ready.

In the inverted pendulum case, there is a little overhead for setting up the necessary persistent storage and the signal handler in the very first execution in the system initialization phase. The time-consuming memory allocation for persistent storage is not present in any subsequent resurrection. After the initialization, each resurrected process will simply attach itself to the already existing storage using one simple system call.

3.4 Discussion

In this chapter, we presented *Process Resurrection*, an efficient recovery mechanism for real-time embedded systems. The time between fault detection and the recovery initiation is very short, and recovery overhead is only incurred when there actually is a crash failure. This ensures that there is minimum overhead for recovery and the remaining processing power can be used for soft or non real-time applications.

We have applied this technique to the inverted pendulum control to show how quickly it can recover from crash failures occurring at a very high rate. The recovery was completed within 12 ms for the control process within a 20 ms minute period on our experimental platform. When implemented on a faster processor like Intel Pentium III 1200 MHz, the whole recovery took less than 700 μs.

When the timing budget is tight or the WCET of recovery is uncertain, various dependency strength weakening methods can be used to extend the deadline. In our example, the Kalman filter and the isolated buffer were used to reduce the disturbance caused by the recovery actions.

It can also be used with other complementary techniques like software rejuvenation [41, 86, 21] and checkpointing and logging [40]. Memory-based persistent storage eliminates the overhead from accessing the disk-based persistent storage. This memory region can easily be remapped at a low cost after resurrection. The use of memory-based persistent storage was demonstrated in the inverted pendulum control experiment. The storage can also be shared by many processes or managed by external programs.

For some applications in a distributed environment, a transparent fast recovery may be desirable. Such applications will be further benefited if a

restart-transparent communication mechanism is provided. Although *Process Resurrection* currently leaves the communication channel recovery to applications, the future plan includes support for such a mechanism in the form of a library and API. Nevertheless, users will be allowed to use other means of communication and design their own recovery mechanisms for the channel.

4 Expanding failure detection coverage with eSimplex

eSimplex is a toolkit developed by this author for implementing analytic redundancy on embedded real-time systems. Use of analytic redundancy allows detection of content errors if a proper safety test can be designed for the application. Since the check is performed at regular interval, timing errors can also be detected.

Our goal is to meet the following requirements:

- Runtime efficiency. Real-time embedded systems are often limited in computational and storage resources. It is unrealistic to have a mechanism that incurs huge runtime computation and storage overhead;
- Predictability. All activities of the system, including active components and support systems, must be predictable, and their WCETs must be known in order to guarantee their schedulability;
- Robustness. Updates must be dependable and should not violate runtime requirements in terms of timing, execution, and semantics. If any anomaly is detected, the system must be recovered in a predictable

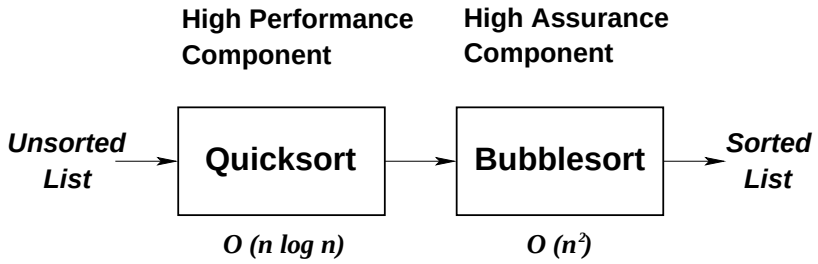

Figure 4.1: Conceptual overview of Simplex

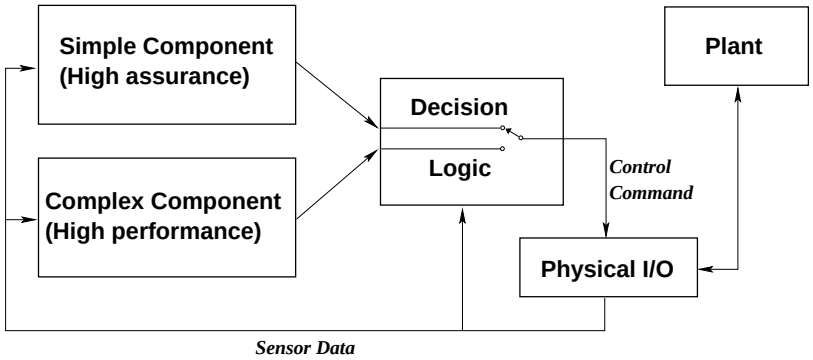

Figure 4.2: The Simplex Architecture

manner;

- Applicability. The majority of embedded systems are still developed in C/C++. Any dependability testing or upgrade system must be compatible with the commonly used development environment.

4.1 Architecture

eSimplex is based on Simplex architecture [79]. Simplex architecture can detect and tolerate timing, execution and content errors of user-supplied software. Figure 4.1 is a conceptual illustration of the principle of this architecture. In this example, we are developing a sorting program. The critical requirement is to sort the list in the correct order, and the optimization requirement is to sort efficiently. We have an implementation of bubble sort, which is logically simple and easier to implement, but the computational complexity is $O(n^2)$. Suppose we can implement, with less confidence, a faster but more complex sorting such as quick sort, whose computation complexity is $O(nlogn)$.

If we choose one implementation over the other, we would either lose performance or accuracy. To do it in Simplex style, we first run the "newly

developed" quick sort implementation, and then pass the results to the Bubble Sort. If the list is sorted correctly by the new sorting program, bubble sort will traverse the list in $O(n)$ time, making the overall complexity $O(nlogn)$, otherwise it will correct the result in $O(n^2)$ time. In this way, we can benefit from the features of the newly developed unproven component when it works correctly, but still meet the critical requirement even if it does not, at the cost of a degraded feature or performance. The key concepts illustrated by this example are the notions of *depend* and *use*. The sorting system's correctness depends on the bubble sort. That is, if the bubble sort does not sort correctly, it will produce an incorrect order. But the system only uses the output of the unproven new sorting algorithm. The system will output the correct order, even if the unproven new sorting algorithm produces an incorrectly ordered list.

Figure 4.2 shows an outline of the actual architecture for real-time embedded systems. In this architecture, a simple and verifiable core component (e.g. bubble sort) performs the critical tasks. A feature rich but unreliable counterpart of the simple component is called the complex component (e.g. quick sort). The complex component may have not been fully tested or verified, and developers can submit one for on-line testing or dependable upgrade. The physical I/O subsystem supplies input data (e.g. sensor readings) to the simple and complex component. The heart of this architecture is the decision logic, which checks not only the outputs from the complex component, but, more importantly, whether the state of the plant stays within the recovery region of the simple controller. This is because it is often impossible to determine if an output from the complex component is correct without examining how the system responds to the output. The system is normally controlled by the complex component. The simple component will take over

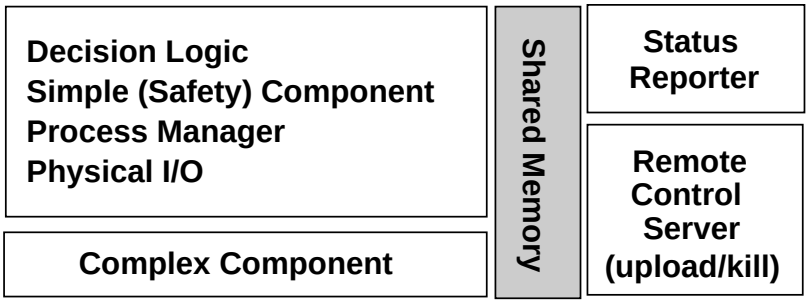

Figure 4.3: eSimplex: An implementation of Simplex for Embedded systems

if 1) the complex component generates an illegal output; 2) the system under the control of complex controller performs poorly; 3) the system state under the complex component approaches the boundary of the recovery region; or 4) the process that runs the complex component fails. In the following discussion, all but the complex components are called system components.

The keys to ensuring dependability of the system are:

- Isolation of the unproven complex component from the system components. By isolation we mean that the unproven complex component is in a separate partition, in time and space. For example, the misbehaving complex component cannot exceed its execution budget or preempt core components. It is also contained in a separate address space to prevent corruption of core components' memory content.

- The system components do not *depend* on the complex component, but *use* its output if available and acceptable based on the decision logic as described above.

4.2 Implementation of the Architecture

The architecture described in the previous section has been implemented to prove its feasibility and to evaluate its real-time properties. For a dependable

testing and upgrade subsystem, we chose to use eSimplex and modify it to suit our purposes. eSimplex is an implementation of the Simplex architecture for embedded systems. It was developed for the Linux operating system, but can easily be ported to any POSIX-compliant Unix-like operating system. Figure 4.3 shows the internals of eSimplex. The system core components and the complex component reside in separate virtual memory address spaces. Soft real-time, non-critical components such as status reporter and remote control server also run as separate processes.

All communications between the processes are performed via the shared memory region with the communication wrapper functions. The use of shared memory has distinctive benefits in real-time embedded systems:

- Shared memory is allocated from the kernel memory, thus it can act as a stable storage as long as the operating system does not fail;
- Once mapped to the local virtual address space, accessing shared memory does not involve any system call, unlike other IPC mechanisms. Since it can be accessed in the application context, no context switching or memory copying is necessary for communication.

eSimplex enforces the *Use* relationship in the application semantics domain, and with the help of the operating system's fixed priority scheduler, isolation in the timing domain is also provided. Isolation in the execution domain is achieved by the operating system's process-level protection and the use of static compiler checks for memory safety [49].

Component	Role	Characteristics
Simple Component	Safe high assurance component.	Real-time and critical
Decision Logic	Check for safety and detect content errors of complex component.	Real-time and critical
Process Manager	Start or stop complex component	Real-time
Physical I/O	Send and receive data from actuators and sensors	Real-time and critical
Complex Component	High performance component submitted by users	Real-time and non-critical Potentially buggy
Status Reporter	Reports plant status and control commands to external user interface	Soft real-time Non-critical
Remote Control Server	Allows complex component to be uploaded. It can also remove complex component per users request.	Non real-time Non-critical

Table 4.1: Description of eSimplex components

4.3 Experiments

For evaluation and analysis, we have implemented an inverted pendulum control system on our architecture.

4.3.1 Inverted pendulum control system

The hardware platform for the control computer has an AMD Élan SC520 embedded processor packaged in the PC/104 form factor. The second layer of the system is a D/A converter for control command output to the actuator. The last layer consists of a custom-made PC/104 sensor interface board for optical quadrature encoders. The system has 64 MB of main memory, but the operating system is allowed to use only 2MB, and 8MB is used for the RAM disk. The system also has 16MB of flash memory for storing the operating system kernel and the root file system image. For timing measurement, since the processor lacks an on-chip cycle counter, an i8254-based timer on the DA converter board was used. The operating system is Linux 2.2.18 with real-time enhancements.

The inverted pendulum control is a periodic task running with a period of 20 ms. When the system is in the testing and upgrade mode, the physical I/O subsystem reads the sensor data for angle and track position. The input data is used by the simple controller (or reliable controller), then the complex controller to calculate the control command, which is the voltage to be sent to the motor moving the cart. Before the deadline (20 ms after the beginning of this period), decision logic performs an acceptance test on the results from the complex controller. If the command is not acceptable, the complex controller is no longer allowed to run. Details about the decision logic are given in the next subsection.

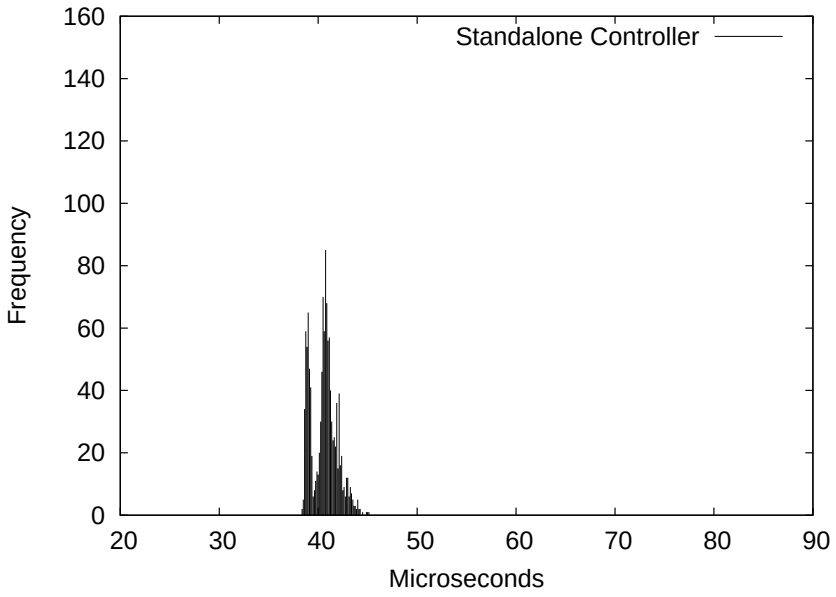

Figure 4.4: Standalone controller with physical I/O. (avg: 40.62 μs, stdev: 1.29, max: 45.13 μs)

4.3.2 Decision logic for inverted pendulum control

Decision logic is where the check against complex controller's content errors is performed. We can define an acceptable system state with a set of operational constraints. The decision logic will take away the control from a faulty complex controller and give it to the simple controller when the plant state approaches the boundary of recovery region. The recovery region is the largest stability envelope for the associated controller within the state constraints. Mathematically, it is an n-dimensional ellipsoid represented by a Lyapunov function. If the current state is inside the ellipsoid, the system will stay in it and converge to the set-point. For the inverted pendulum, we define the set-point to be an upright pendulum at the center of the track. LMI tools [12] are used to find the largest ellipsoid to give the maximum freedom to the complex controller being tested. In reality a slightly smaller ellipsoid is used to guard against errors in the model, actuation and sensing.

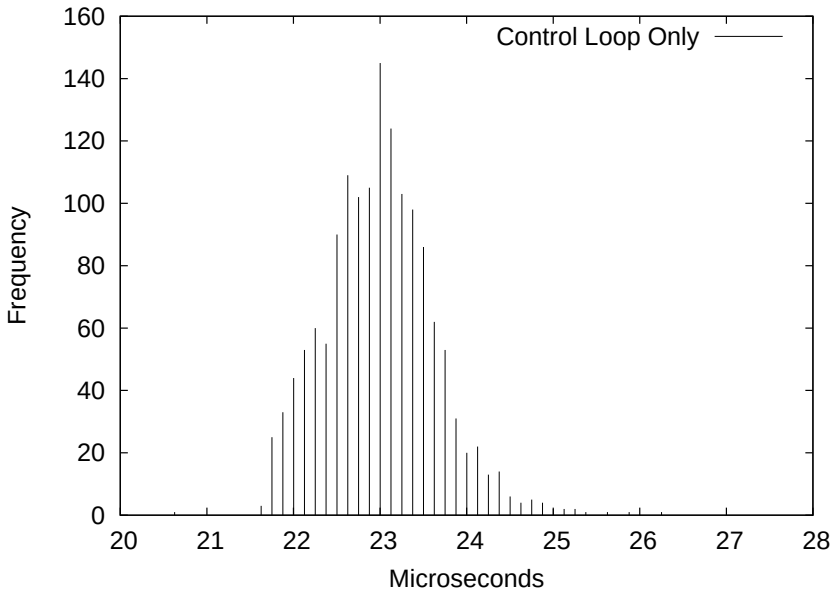

Figure 4.5: Control loop only. (avg: 23.0 μs, stdev: 0.64, max: 26.25 μs)

This check is performed during every sampling period when the complex controller is submitted and running. When the complex controller fails the check, eSimplex kills it, and the control command from the simple controller is used instead, in subsequent periods, until another complex controller is submitted.

It is important to note that the safety check is for predicting and preventing unsafe state transitions. Focusing on the symptoms and effects of errors, rather than errors and faults, has a good reason. Some types of errors are impossible to catch immediately. For example, suppose that we have a uniform random number generator. Given the first three of numbers generated, say, 0.15, 0.35, and 0.15, there is no way for sure to tell whether it is faulty or correct. Instead, cumulative effects and symptoms are monitored, and if any potential to deviate from the specification is found, a flag is raised.

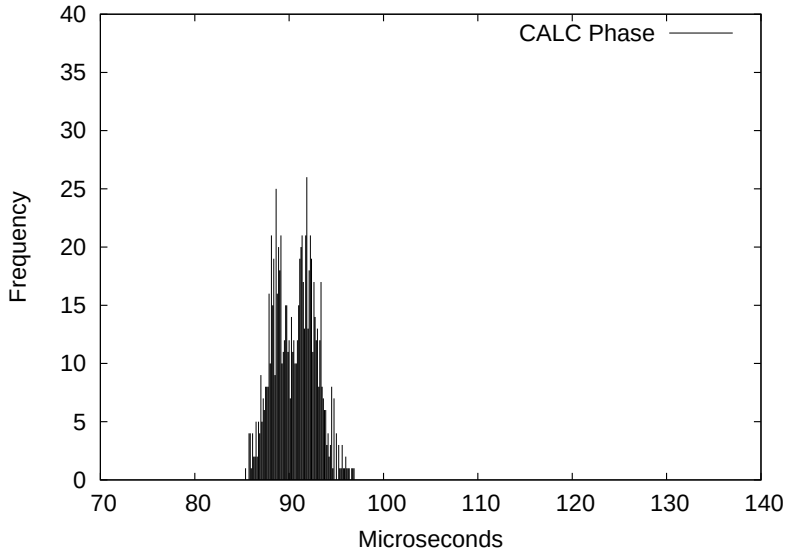

Figure 4.6: CALC phase execution times, avg: 90.6μs, stdev: 2.24, max: 96.88μs

4.4 Measurements and Evaluation

4.4.1 Timing

For evaluation purposes, various temporal characteristics have been measured. First the execution of eSimplex is divided into two phases and measured, respectively. The CALC phase includes sensing, simple and reliable control, and processing of any subscription request by the complex controller. After this phase, a complex controller is allowed to run. What follows is the OUTPUT phase, which includes LMI-based decision logic and actuation. Figure 4.6 and 4.7 show the execution time distributions.

The total execution time of eSimplex is 176.64 μs on average. The worst case was 191.51 μs. In order to measure the overhead of eSimplex-based redundant control and monitoring, we measured the execution time of the standalone control with the same control code and the physical I/O module.

The standalone version of the controller's average execution time was

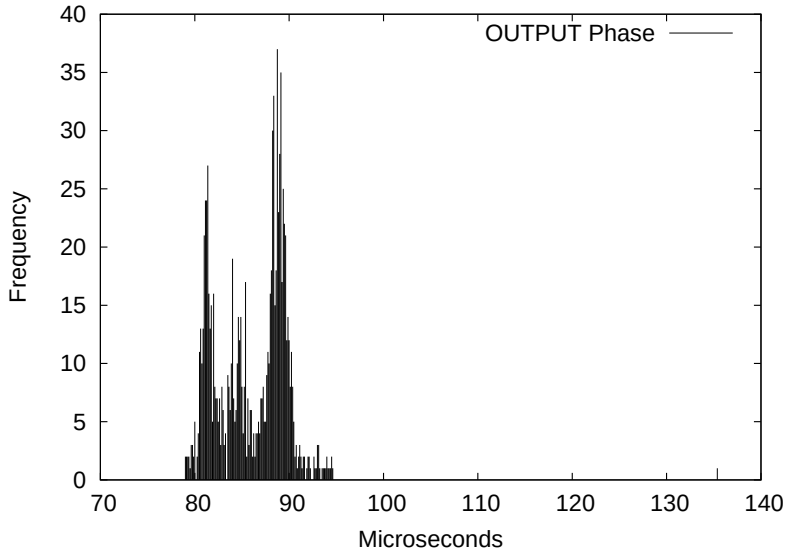

Figure 4.7: OUTPUT phase execution times, avg: 86.04μs, stdev: 3.58, max: 94.63μs)

40.62 μs, and thus the overhead of eSimplex monitoring is 136.02 μs. According to additional measurements shown in Figure 4.5 and 4.4, the physical I/O takes about 17 μs. This is mostly from the sensor reading code, which requires ten port I/O operations, each of which takes about 1 μs. This code also contains conditional extra processing for negative values when assembling the values read from the register. The effect is shown in Figure 4.6 (a)as two peaks, one for positive and the other for negative angle readings.

Out of 136.02 μs of overhead from eSimplex, the amount incurred by CALC phase housekeeping operations (approximately 90.6 - 40.62 = 50 μs) will stay more or less constant, even if a much more time-consuming application is used. As a result, when the system is scaled up, the dominant factor of overhead will be the application-specific algorithm used in the decision logic.

Software bugs	Type of errors
Infinite loop	Timing, omission
Segmentation fault	Timing, omission
Divide by zero	Timing, omission
Sudden max	Content
Gradual push	Content
Output 0.0 V	Content

Table 4.2: Types of injected faults

4.4.2 Detection and recovery

The tests on how well our experimental application detects and recovers from errors, we have injected many different faults at random time instances. Table 4.2 summarizes the injected faults.

After tuning the *safety threshold value* in the decision module, no content error could make the pendulum to fall. Dealing with various fail-silent modes were simpler. Crash failures were safely translated to contained fail-stop thanks to the process-level protection provided by the operating system. But if the process-level protection is bypassed using system calls such as `kill( )`, the whole system can be compromised. Thus this degree of integration should be allowed only when the criticality of all redundant components are the same.

4.5 Discussion

In this paper, we described an architecture for dependable on-line testing and upgrade for real-time embedded systems. We implemented the architecture with an embedded version of Simplex and used Process Resurrection for mode switching. Let us reexamine the requirements listed in the introduction to see how the findings contribute to our overall goals.

Runtime efficiency. According to the measurements, the fixed part of dependable testing and upgrade environment only incurs about 50 μs. When the execution of a simple PID controller takes 40.62 μs, this is not a significant overhead. The major source of the overhead is in the decision logic, which is application-specific. The mode switching is also very efficient. When staying in one mode, there is no computational overhead and only tens of bytes of storage overhead. Users can choose to remove the monitoring and redundant execution by mode switching until another testing and upgrade is needed. In normal operation mode, no computational overhead exists.

Predictability. The variations in all measured execution times are very small compared to the period of controllers in our experiment. The predictability can be further improved by using a more predictable operating system. In our case, Linux met the requirements. For boundary WCET conditions in control systems, a timing buffer such as the Kalman filter can be used.

Robustness. In our testing and upgrade mode, the content errors of applications can be also checked by on-line monitoring, and recovered in real-time by use of the analytically redundant component in eSimplex.

Applicability. The whole system is implemented in C, and no proprietary modification is done to the operating system kernel. Any POSIX-compliant Unix-like operating system can use our framework.

The complete coverage of content errors (i.e. unacceptable output) in control commands is guaranteed by the safety check inside the decision logic based on LMI calculations. In addition to the safety check, there are number of checks including a simple voltage range (-5 to +5V) check and a data validity check based on finite() in the standard math library. Without the latter check, some comparisons can return unintended results.

The safety check is mathematically proven to guarantee the system to detect the faulty control command that will make the system unrecoverable. In order to verify that the implementation actually works, we inserted various value faults in the complex controller. One of the most elaborate was designed by a control engineer. It swings the pendulum with an extra force at near the both ends of track. In the normal operation mode without the safety check, the pendulum would crash. In the testing and upgrade mode, in which eSimplex is enabled, the complex controller with the artificial faults were eventually rejected, preventing the pendulum from falling.

One weakness of the tightly coupled eSimplex architecture is that a non-critical component capable of killing all in the process group can shut down eSimplex. In VEER, we solve this problem with the partition and protection provided by RT-VMM

```
{
      static float wdither = 0.0;
      static int wratio = 1;
      static float wincremental = 0.30;

      glob_ctr+=2;

      if (( glob_ctr % 2000 ) == 0 )
      {
         /* wratio = 2; */
         wincremental += 0.30;
         wdither = wincremental;
      }

      if (( glob_ctr % 100 ) == 0 )
      {
         wratio += 10;
      }

      if ((glob_ctr % wratio) == 0)
      {
         if ( wdither == wincremental )
            wdither = -wincremental;
         else if ( wdither == -wincremental )
            wdither = wincremental;

      }
      command += wdither;
   }
```

Table 4.3: Sample faulty control code: "gradual push", generating content error

5 The real-time virtual machine monitor

5.1 Background

The concept of virtual machine began to emerge in early to mid-1960s. The first commercial product that incorporated virtual machine support was IBM S/360 model 67 and it was announced in 1965. The accompanying virtual machine control software, CP-67 became available in 1967. It was very successful and won over time sharing operating systems such as TSS/360 and Multics. But perhaps most well-known product is IBM S/370, which included the popular feature by default [65].

As desktop and workstation computers got powerful, the use of virtual machine gained its popularity outside the mainframe computing. One of the first product was VMWare in 1999. Since then, many commercial and open-source virtualization products have been released. Java's success also application-level virtualization popular. But we will focus on hardware-level virtualization.

Hardware-level virtualization allows multiple operating systems to run at the same time. In order to efficiently support hardware-level virtualization, certain conditions must be met. Unfortunately, Intel x86 line of processors until recently only provided VM86 mode for virtualized 8086. Intel x86 processors did not satisfy certain conditions in Popek and Goldberg's formal requirement [69, 75]. Because of this deficiency, full native virtualization was impossible. Some used emulation or binary translation (e.g. qemu, VMWare,

and VirtualPC), the others used paravirtualization (Xen and Denali). While emulation and binary translation provides virtual environment that is identical to the native one, the performance often is an issue due to extra works needed for the translation. Paravirtualization slightly modifies the machine architecture to achieve a better performance, but it loses binary compatibility.

For building RT-VMM on commonly available traditional x86 architecture (pre-VT), we chose paravirtualization for performance and timing reasons. The basis for our implementation is Xen 2.0.7 [9], since the other mature open-source paravirtualizing VMM, Denali [87] required recompilation of applications due to its ABI modifications. Many of the new Intel x86 processors as of 2006 are equipped with Intel Virtualization Technology (VT), promising the full native virtualization without any need for emulation. However, as of writing, there is no product, open source or commercial, that is stably supporting VT [43]. In fact, the hardware specification is still evolving; Intel has revised the specification and also added VT-d for I/O virtualization in mid-2006.

Real-time virtual machine (RTVM) [81] concept has been suggested for migrating and consolidating real-time embedded systems. In avionics applications, subsystem consolidation has been popular with the development of the integrated modular avionics or IMA [3, 76]. FAA has embraced the concept and also approved the use of RTVM that enables multiple cyclic executives or rate monotonically scheduled legacy subsystems to be run on a single hardware without going through costly recertification processes.

5.2 Desired Properties of RTVM

5.2.1 Protection and isolation

No protection

A single address space operating systems have been widely used for simple and low-power applications on equally less capable processors. For most embedded systems, cost has been the main reason why commercial manufacturers choose such a platform. But safety-critical real-time embedded systems developers have favored it because its simplicity yields better predictability. For these types of systems, comprehensive code analysis is required to be certified by the standards such as FAA's DO-178B level A. Since the high cost offline verification and testing is performed for certification, protection is also often provided in the form of a logical protection, such as the one described in ARINC 651-1, "Design Guidance for Integrated Modular Avionics" [28].

For general purpose real-time embedded systems, such an expensive verification process cannot be justified and it will be prohibitively costly to perform a comparable degree of checks on much more complex software systems. Moreover, statically designed, analyzed, composed systems are not suitable for runtime system configuration changes.

Process-level protection

The most modern general purpose operating systems offer process-level protection utilizing the memory management unit, context switching support, and different privilege levels. As part of an operating system, system calls are provided to user mode processes. System calls are executed in the kernel mode (i.e. privileged mode) and some of them can affect other processes in

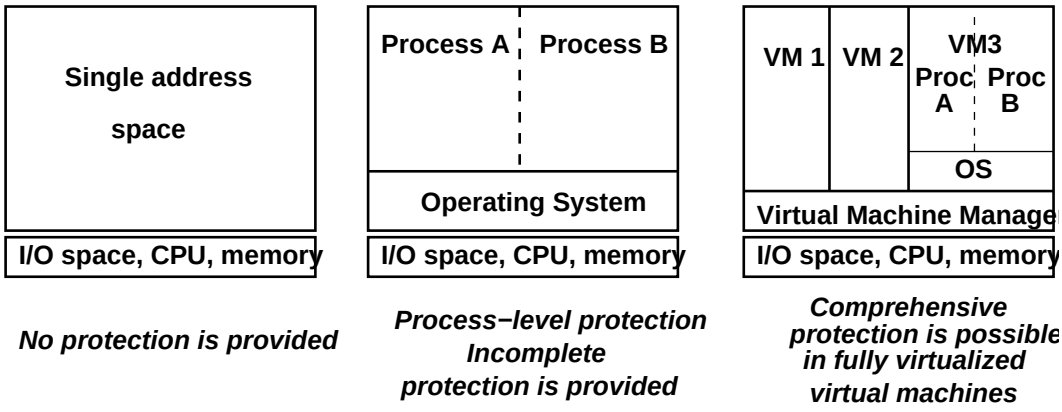

Figure 5.1: Different protection levels

adverse and unexpected ways [48].

The focus of spacial protection is on main memory, and other accessible spaces are not well protected or partitionable. For example, the Linux protection mechanism on the x86 I/O port address space is not as comprehensive or sophisticated as its virtual memory implementation. This is because hardware access is supposed to be done through the device driver interfaces exported in file systems. The standard user-level access control mechanism of Unix operating system is also applied to device driver interfaces.

In addition to the limitations in spacial protection and partitioning, process-level protection traditionally involves relatively large operating system kernel. It is almost impossible for large kernel to get high level safety certification. The complex structure and features have more potential to have residual bugs than small and simple kernels. The complex kernel and task management also implies longer context switching times.

VM-based protection

Virtual machines provide the strongest protection and partitioning of time and space. The relationship between a virtual machine architecture and the

actual hardware architecture it is running on determines many properties of the virtual machine. Here, by virtual machine we mean the systems in which virtual machine monitors communicate with hardware directly and manage guest operating systems in each virtual machine.

Real-time embedded systems can also benefit from VM environment. For distributed systems with slow processors, a powerful modern processor with a proper VM environment can reduce the hardware and maintenance cost. The isolation provided by a VM architecture can be designed to match the existing hardware partitioning while providing better collaborative environment between VMs. This, however, cannot be achieved by existing non-real-time VM architectures.

When multiple software components are placed in different virtual machines according to their criticality levels, an ideal real-time virtual machine manager will assign predefined time slots to each virtual machines in predictable manner. Since the virtual machine monitor maintains very little state, switching between virtual machines can be simpler and quicker. Predictable resource allocation also enables us to assign extra resources to a specific virtual machine while not affecting other critical virtual machines. It is only possible by marriage of strongly partitioned and virtualized runtime environment and the characteristics of real-time computing. This issue will be further discussed along with the description of recovery mechanisms.

5.2.2 Resource virtualization

Resource in a computer system is defined as anything that software can use or occupy to accomplish its desired tasks. The most important resource is, of course, the CPU. A *pure* virtualization will expose the same CPU architecture including the instruction set and registers to its guests. In ideal case,

the only difference seen by guests will be the speed of execution. Similarly, the virtual machines memory architecture must appear identical to that of real hardware. I/O space and interrupts also need to be partitioned, masked, multiplexed or translated in order to create a virtually identical structure and organization.

Resource virtualization is realized in different ways and at different levels of completeness due to different goals of each design. On some computer architecture, complete virtualization requires a high overhead, therefore forcing the designers to make a trade-off between efficiency and completeness.

By providing a virtualized hardware architecture that is structurally and semantically identical to the actual hardware, the existing software including operating systems and applications can run without any modification. This property is especially important if the software cannot be modified due to lack of source code access. Pure virtualization is also required to support efficient online migrations between physical and virtual environment. This complete virtualization could be costly on some architectures with incomplete provisions for virtual machine operation.

Paravirtualization is one way of mitigating the performance overhead. Instead of performing binary-translation for the unprotected subset of privileged instructions, by providing an optimized interface that is different from the ISA, the expensive operations can be replaced. The downside is, of course, the exported virtual machine architecture is not identical to the hardware, requiring modifications to the existing operating systems.

Intel Pentium processor, the target processor architecture of our project, has limitation that makes complete virtualization expensive [75]. This is why we choose paravirtualization.

5.2.3 Temporal properties

An RTVM architecture, being a real-time system itself, must designed to exhibit predictable timing behavior. In the heart of the architecture is RT-VMM , which must be able to switch between RTVMs in bounded time according to a schedule.

In addition to RTVM switching, executions of the hypercalls implemented in RTVM monitors should be kept short and predictable. If an RTVM causes hypercalls to be executed by invoking privileged instructions, the calls must execute to completion even if the CPU budget for the RTVM is depleted. This is because the state of RTVM monitors must be kept consistent when switching to a different RTVM. In this case, we must be able to tell what is the maximum possible overtime that an RTVM can have due to a hypercall.

Although the ideal RTVM should provide an environment equivalent to the physical CPU at 1/n speed, it is extremely hard to closely emulate such an environment. If an RTVM is given 1 second slot every 10 seconds, the effective processing power will be 1/10 of the physical processor. However, if the original software written for a processor with 1/10 performance and was designed to send a message at the end of its period of 10 seconds, the timing will be off in the RTVM by sending it 9 seconds early. In traditional cyclical executive systems, such timing is fixed by design and other cooperating systems are built on that knowledge. By further dividing the size of RTVM execution slots or quantum, this effect can be reduced, but this also increases the VM switching overhead. Thus when mixing legacy cyclical executive tasks and new priority scheduled tasks, a timing buffer may be necessary.

5.2.4 Spacial properties

It is vital to maintain the ability to switch RTVMs instantly whenever necessary. Any work in spacial dimension must be preemptible or have very short predictable preemption delay. In addition to the preemtibility, VM switching should not entail too much bookeeping. Since the accurate timing is the most important property we want to have in real-time systems, the rule of trade-off in spacial dimension changes from non-real-time systems. For example, Intel i386 processor does not support virtualization of its port I/O address space in protected mode. For compatibility reasons, some commercial VMMs virtualize the space in software though binary translation [84], but the overhead of virtualization may not be acceptable for RTVM monitors.

A more formal description on required properties are given in [69].

5.2.5 Instruction virtualization

At processor instruction level, all privileged instructions are emulated by RTVM monitor code. When a privileged instruction (e.g. HLT) is executed from a virtual machine, a CPU trap will let the RTVM monitor to handle the situation by simulating the effect of the instruction invocation for the virtual machine in a safe way.

The privileged instructions are as follows:

- HLT, CLTS, LGDT, LIDT, LLDT, LTR, LMSW, and MOV (to/from control/debug/test registers) for i386
- INVD, WBINVD, and INVLPG for i486
- RDMSR, RDTSC, and WRMSR for Pentium and later.

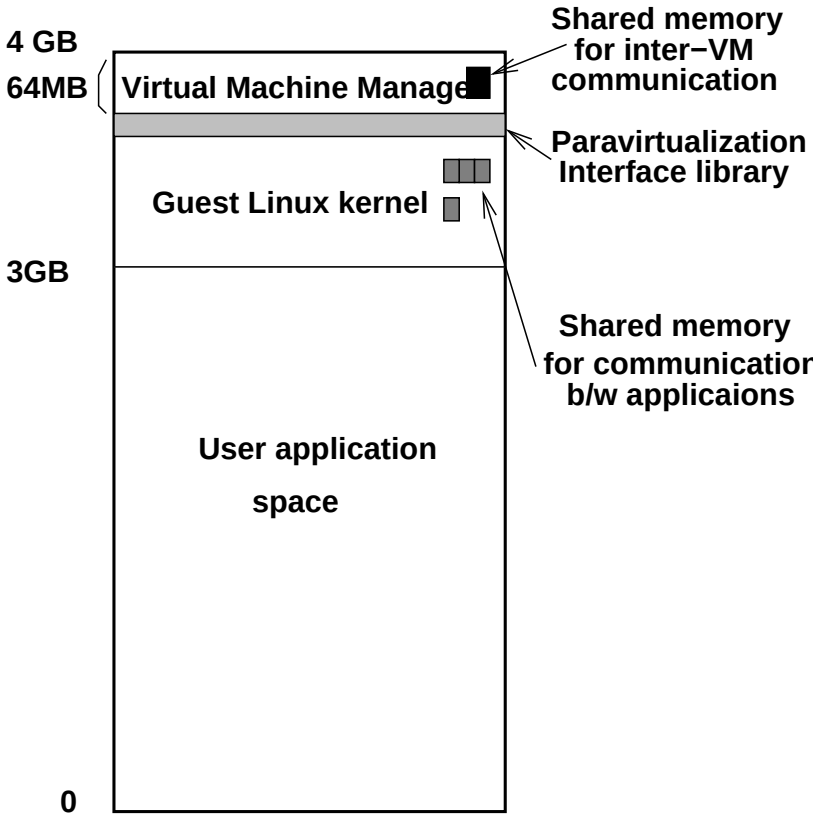

Figure 5.2: The virtual memory space map of a virtual machine

The system must enforce the temporal and spacial partitioning, instead of relying on voluntary conformance.

5.3 Enforcing partitioning and fault containment

The RTVM architecture [81] for avionics systems imposes a set of rules that the guest software must follow. Since these systems are tested and verified throughly for the certification, the conformance to the system rules may not be strictly enforced by run-time environments. For general real-time applications with potentially more bugs, the isolation and partitioning of

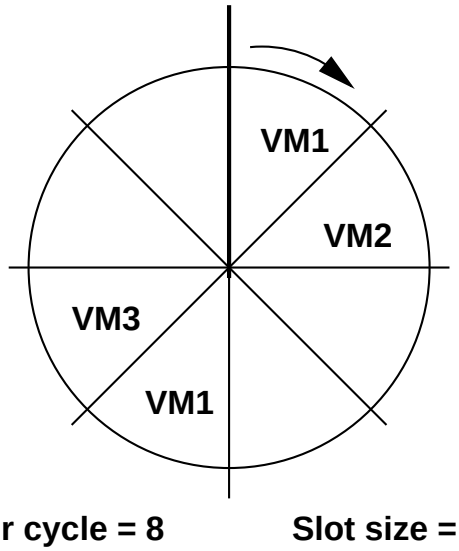

Figure 5.3: A conceptual view of the RTVM scheduler

resources must be enforced in the system level, like the conventional virtual machine monitor-based (VMM) execution environments. This is why we need a real-time VMM as the foundation of VEER.

5.3.1 Why partition

In order for an RTVM system to work properly, component faults must be isolated with no hidden path of propagation. The partitioning can be obtained by using special offline analysis and design tools, but the cost of reaching a high coverage for complex real-time systems can be too high. Also, it is only suitable for statically scheduled systems, not for dynamic systems.

One could use a conventional operating system as an isolation environment. It is true that the process-level protection does provide a string protection in memory space, and also use of real-time scheduler can provide a temporal partitioning. However, since other operating system features are shared, one process can affect another in various ways. For example, if multiple processes belong to one process group, a kill system call by one process

can terminate all others. Also, since system calls are not guaranteed to be idempotent, sharing of kernel features can lead to inconsistent and unpredictable states. Commercial safety-critical operating systems are designed with this issue in mind, and application and OS code is analyzed and verified so that a preemption always leaves the OS in a consistent state. This kind of verification is almost impossible for general purpose operating systems.

The strong spatial and temporal partitioning and the ability to dynamically and accurately reconfigure the partitioning are the key enabling factors of real-time virtual machines.

5.3.2 Timing and spatial partitioning

For real-time systems, temporal partitioning is the most unique and important among many others. In order to have a strict control over timings from the RT-VMM to applications in VMs, the RT-VMM's periodic timer event must be consistently propagated down to guest kernels. The semantics of timer events must be also consistent across the system.

In addition to temporal partitioning, spacial partitioning also plays an important role in real-time systems. If a failure occurs in a component, it is desirable to prevent it from propagate to other components. Since the RT-VMM provides hardware-level virtualization, careful organization of components will enable the fault containment down to the lowest level. For example, a mismanagement of video card by a UI component may make the console unusable and require system reset to recover. With RT-VMM, the UI handling subsystem with hardware access can be isolated in a separate VM, so that the recovery of UI can independently performed without disturbing other components. Containing faults also helps reducing the recovery time.

In normal operation mode, the RT-VMM schedules VMs in a predeter-

mined fixed time slot according to the scheduling method illustrated in [81], which is used in some of the DO 178B level A certified systems (Figure 5.3). The major cycle time must be no longer than the minimum of all task periods in the system. In theory, the size of the time slot should ideally be the greatest common divisor of the WCETs. But in reality smaller time slots will incur more scheduling overhead. The experimental result section shows the amount of overhead we can expect.

5.4 The RT-VMM architecture

5.4.1 Task scheduler

The task scheduler is responsible for scheduling both normal case executions and recovery actions of tasks.

If we are to use conventional schedulability analysis based on the worst case recovery times of all tasks, its normal case CPU utilization will be very low. For a reasonably reliable system, it is a waste of CPU times. It also cannot distinguish between critical and non-critical recovery. If the scheduler is modified to deal with all cases, it will be a very complex core component with a more probability of having bugs.

To make scheduling simpler and controllable, we utilize a virtual machine architecture to divide scheduler into two levels. The top-level scheduler in the virtual machine monitor is based on a simple TDM-like scheduler, which assigns one or more slots to each real-time virtual machine. It has been shown that the combination of this high-level TDM-like scheduler and intra-VM real-time priority scheduler can work in predictable way with computable schedulability bounds [81].

5.4.2 Event handler/schedulers

Interrupts are asynchronous events. When an interrupt is raised, an interrupt service routine is executed to process the event. Unless it is disabled in the hardware (i.e. programmable interrupt controller), the software has no control over how often interrupts can occur. If an interrupt line is disabled arbitrarily, the system may miss an important event and loose data. If we allow free interruption of real-time tasks, the blocking times will be too long.

In a virtual machine environment, it can be worse. Suppose a virtual machine is getting time slots worth of 10 % of total CPU time. The ideal virtual machine would simulate a slow machine (1/10 speed). However, if interrupt is generated too frequently, it can exceed the maximum rate that can be generated on the simulated slow machine, and introduce much more frequent blocking to the software running on the virtual machine.

For this reason, asynchronous events must be regulated in a special way so that the blocking times are reasonably short and bounded, and the probability of missing an event is low.

5.4.3 Control VM

After the system is booted with the RT-VMM, virtual machines need to be started. In a static system, since the organization and configuration virtual machines do not change, a static boot-time start-up scenario works well. But for the RT-VMM to work, we need a capability to start, stop, and change configuration of virtual machines. This, obviously cannot be done with a fixed start-up script.

One of simple approaches is building all VM management functions in the RT-VMM. This would work only if the RT-VMM has all of high-level

features implemented internally. For example, if saving configuration data to a disk is needed, the RTVM monitor needs a device driver for the disk and a file system driver. Also, if disk write is issued by the most privileged RT-VMM, all virtual machines running on top of it will see the I/O time as a blocking time. If it is done by a virtual machine, it will be a part of its execution time or a local blocking time.

For this reason, a user level proxy is used. The proxy runs inside the control VM, a privileged to have access to hypercalls in the RT-VMM. The proxy can start, stop, and submit a new configuration. Whether to honor it or not is entirely upto the RT-VMM. If the monitor finds submitted configuration is not appropriate, it may reject it.

5.4.4 Summary

We have discussed how the RT-VMM architecture is organized and how each element must function. The major challenges are summarized as follows:

- Enforcing a safety in critical ordering with traditional dependency management techniques can be too restrictive. We need to utilize application-specific data usage semantics and enforce Use relations.

- Partitioning based on static analysis/verification or even process-level protection is not adequate. Static methods have limitations in checking certain dynamic properties, and with process-level protections, it is still possible to affect other processes with system calls and by making the OS state inconsistent. Thus we need a stronger partitioning and protection, such as the one provided by a virtual machine architecture

- For the task and recovery scheduler to work properly, modification of existing priority scheduler has limitations. If a simple analysis is done

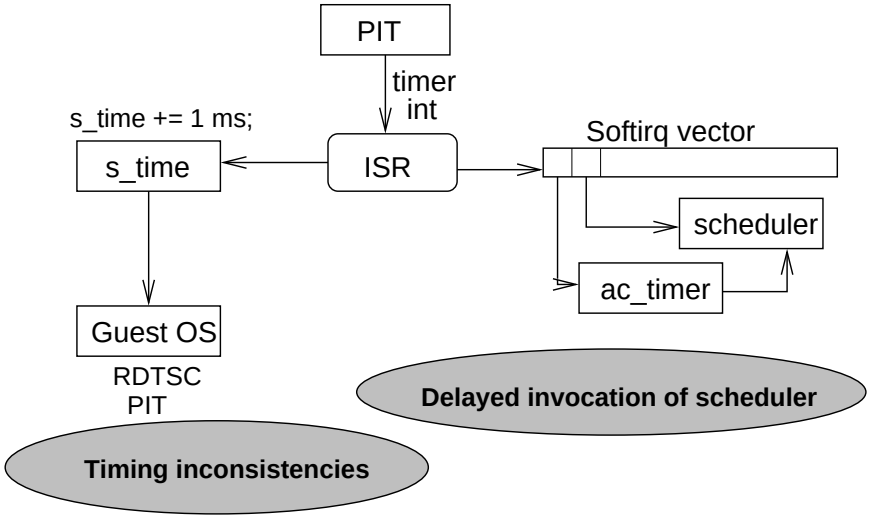

Figure 5.4: Non-real-time nature of the original Xen

without any modification, the outcome will be too pessimistic, making the CPU utilization too low. If the full modification will result in a very complex scheduler which may contain a new bug. We approach this with two-level scheduler with each level implements relatively simple and straight forward scheduling, but becomes powerful when combined.

- Asynchronous events can disturb a virtual machine by generating a burst of interrupts in a short period time. If we allow this, the maximum blocking time for each virtual machine will be too long. We also cannot simply disable interrupts and miss events. We will transform asynchronous events into pseudo-synchronous events with low probability of losing events and a better schedulability bounds for each VM.

5.5 RT-VMM implementation

We implemented the RT-VMM on the Intel x86 architecture. We modified and extended Xen, an open source VMM. Xen 2.0.7 on Intel x86 provides a form paravirtualization that requires the guest operating systems to be

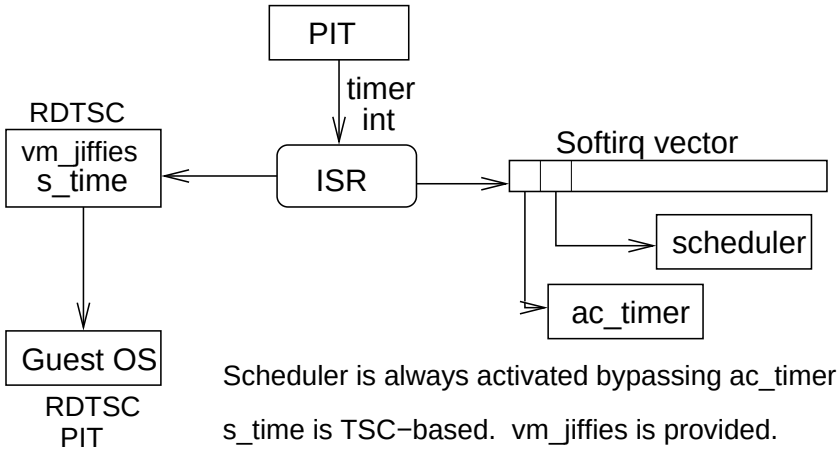

Figure 5.5: Modified for more predictable timings

slightly modified. At the cost of slight incompatibility at the low-level, Xen performs very well compared to binary translation-based approaches. Since our system would benefit more from the performance than the backward compatibility, we chose Xen 2.0.7 over other candidates.

5.5.1 Timing

Since Xen is designed for non-real-time systems, we first had to modify the way it keeps track of time and delivers "ticks" to guest operating systems. In non-real-time systems, a delay or loss of a timer tick may not matter much, but it will most certainly affect the correctness of real-time systems.

The source of problem is two fold. (See Figure 5.4 and 5.5) First, the delivery of the virtual timer interrupt could be delayed. For real-time systems, timer interrupts must be delivered in timely manner with a small jitter. When a VM scheduling event and a delivery of virtual interrupt are not synchronized, a guest operating system may not be able to correctly perform certain time-bound tasks such application scheduling. We modified the softirq for the scheduler to be raised every time a timer interrupt occurs so that the scheduler can run at every hardware timer interrupt whose period

is configured to be the minimum VM schedule quantum. Since the periodic timer interrupt handler does not trail time-consuming tasks when configured appropriately, virtual timer interrupts are delivered with an acceptable amount of delay.

Second, a guest operating system in a VM could use inconsistent means of interpreting the time. For example, Xen's system time is simply incremented by a preset constant approximation (1000/HZ) of the timer interval, while applications and guest operating systems are capable of referring to more accurate time reference such as IA-32 time stamp counter. Since XenLinux uses Xen's system time as its time reference and the time progression is internally determined (e.g. incrementing jiffies), some of the time-bound events (e.g. POSIX interval timer) may not work as expected. For real-time systems, accurate time reference and a consistent method of interpreting time progression are required. We unify the time reference to the TSC. Since the semantics of the timer interrupt is very clear and deterministic in RT-VMM, we propagate VMM's jiffies value to make the time progression uniform across the system.

XenLinux, the guest operating system, needs changes in addition to the ones mentioned above. Most notably, the way timed events are fired needed a modification. Although the stock Linux having "at-least" semantics for timed events may be appropriate for non-real-time applications, it is problematic for real-time applications as the semantics allows deadline misses. Since we have made the "time progression" deterministic across the system, the semantics was changed to "exactly". If the system is designed to keep the non-preemptible kernel section short (e.g. no lengthy serial communication), this semantics can always be honored.

5.5.2 Scheduling and Recovery

The VM scheduling is carried out by a time division multiplexing-like (TDM) scheduler that supports the scheduling paradigm described in [81]. It is a static scheduler that needs offline schedulability tests. After the slot assignment is done, the schedule and other parameters such as scheduling quantum or slot size and the size of a cycle are supplied to the RT-VMM at its build time. The scheduler assigns slots to launched VMs and schedule them according to the plan.

The RT-VMM does not reschedule even if the current VM is idle, nor does the idle handler of our guest Linux invoke the hypercall for rescheduling. This simplifies the scheduler, since the reschedule is only done at the slot boundaries. There is no need to keep track of individual VM's time consumption.

The scheduler must keep track of how the slots are used when increasing the number of allocated slots to a recovering VM. This is called boosting. If a boosting occurs, any free slots are first used. If no free slot is available or more slots are needed, the scheduler selects victims based on the offline recovery plan. The scheduler enters the boost mode if a specific hypercall is made by the control VM, a special VM that monitors the health of other VMs. This hypercall can also be invoked by the VM with failed components. The scheduler exits the boost mode either when an explicit hypercall is made to notify the end of recovery or when the maximum number of extra time slot assignment has reached.

Faults occurs at different layers of abstractions, RTVM only defects faults violating the VM partition rules and VM API usage rules. For this reason, we do not offer a generalized detection mechanism. We provide primitives

and usage guidelines for designing application-specific methods.

5.5.3 Interrupts and Hypercalls

Even though some hypercall interfaces were modified, the original Xen's design is mostly used as is. This is also true for the way RT-VMM handles interrupts. Privileged virtual machines run privileged kernel and provide device drivers and ISR.

The RT-VMM execution modes can be divided into two: the internal and the external mode. In the internal mode, the RT-VMM runs several short bookkeeping and clean-up operations, and most importantly the scheduler. Since these are ran periodically in RT-VMM, they can be viewed by a VM as a very short high priority task. In the external mode, hardware interrupts cause ISRs to run and software service requests are carried out as hypercalls. As hypercalls run on behalf of the guest software, they can be counted as a part of the local VM execution time. Periodic interrupts can be regarded as synchronous events, so that real-time can easily take them into account when performing timing analysis.

Purely asynchronous interrupts require a more careful look. Open network interface is an example. While maximizing the throughput is the goal of the most non-real-time systems, real-time systems prefer limiting the performance if the blocking times are reduced and the system becomes more predictable. We offer users two options: 1) enable the invocation of the ISR in the privileged guest whenever the corresponding interrupt comes in. If the length of the ISR is long or the interrupt rate is very high, the effect on the unrelated VMs may be significant. 2) The RT-VMM can be optionally configured to disable the handling of the interrupt when the corresponding driver VM is not currently running. This option effectively limits the interrupt rate

and minimizes the interrupts' effect on other VMs.

5.5.4 Partitioning and Virtualization

Enforced partitioning provides isolation. Virtualization provides a uniform interface to the underlying resources so that migration or reuse of software is simplified. Since the paravirtualized architecture by Xen is also used in the RT-VMM, the guest operating systems must be modified. For the sake of simplicity we adopted XenLinux, which is based on Linux kernel 2.6.11.

Since we are using Xen 2.0.7's method of partitioning and virtualization, the I/O space is not ideally partitioned or virtualized. For PCI devices, the RT-VMM inherits Xen's capability of partitioning the devices by making only selected devices visible to a specific VM. This, however, does not control the I/O privilege of a privileged VM in detail, so any privileged VM can freely perform any port I/O operations. A very coerce guest kernel-based control over I/O privilege level is provided by not allowing unprivileged VMs to do any port I/O operations. Since direct hardware access is often required by real-time embedded system software, use of binary translation or emulation as in VMware [84] and QEMU [10] is not suitable. It is ideal to have something equivalent to IOMMU for virtualization and access control, but until it is widespread, we resort to limited solutions.

5.6 Experiments and Evaluation

The experimental system is based on an MMC-2 Mobile Pentium III 750 MHz processor with 288MB of SD-RAM at 100MHz and Intel 440BX chipset. The installed peripherals include two video cards (1 built-in), one network, one BT878-based frame grabber and one D/A - A/D card for controlling the

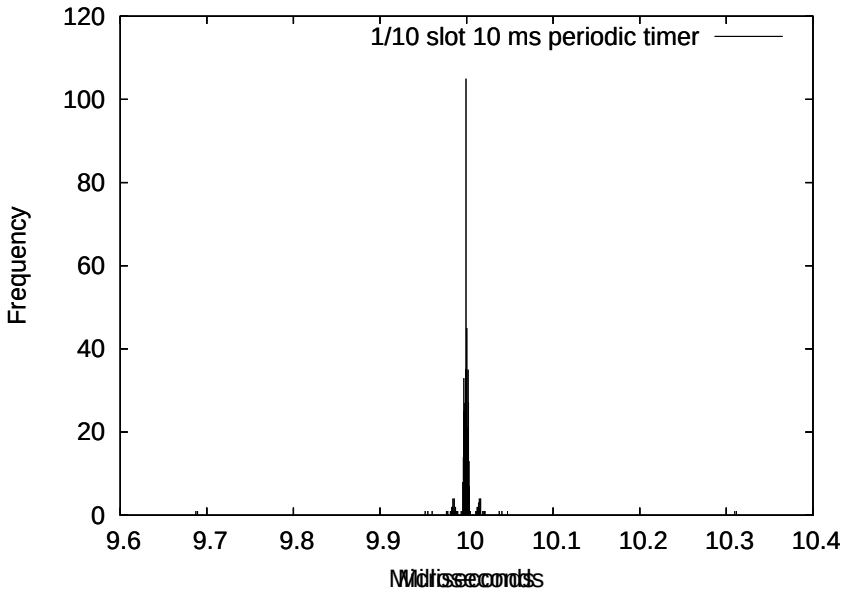

Figure 5.6: Periodic timer accuracy: single 1ms slot (avg:9.999 ms, stdev:0.035, max:10.385 ms, min: 9.614 ms)

inverted pendulum.

5.6.1 Overhead of RT-VMM

Periodic timer

On our experimental system, the major cycle time is 10 ms and the slot size is 1 ms. We first measured the accuracy of the interval timer in the guest Linux operating system. Figure 5.6, 5.7, and 5.8 show the result. When all ten slots were allocated to a single VM, the average interval of 10 ms interval timer was 9.999 ms with 771 μs of variation range. The standard deviation was 0.035. When only one slot is assigned out of ten, the result shows no significant changes. The average was 9.999 ms, and the variation range was 625 μs. The quality of the interval timer depends on both the RT-VMM latency and the guest operating system's maximum blocking time.

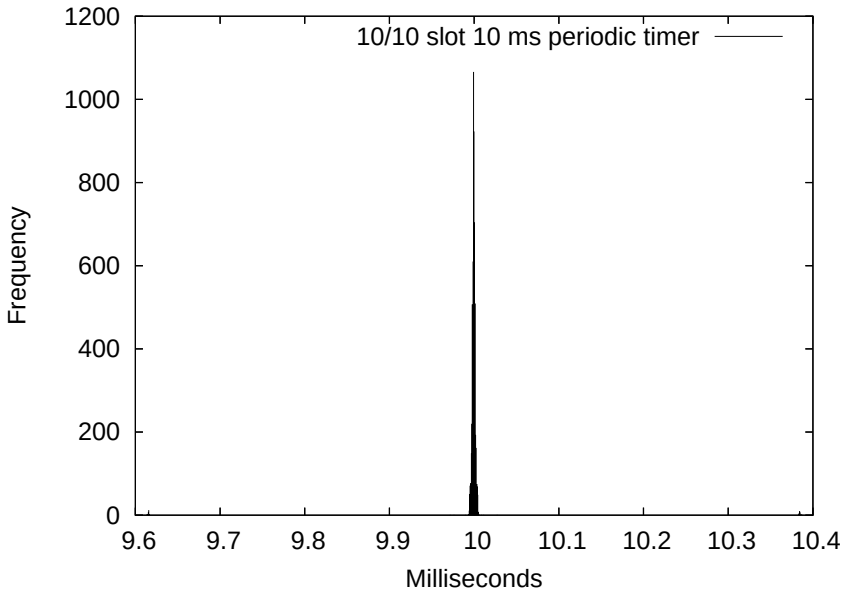

Figure 5.7: Periodic timer accuracy: ten 1ms slots (avg:9.999 ms, stdev:0.029, max:10.312 ms, min:9.687),

Hypercall blocking times

We reused the Xen's hypercall interface with a small change in time-related calls. In this hypercall architecture, `multicall()` combines series of hypercalls. Individual hypercalls made during the steady state after start-up are `mmu_update()`, `update_va_mapping()`, `update_descriptor()`, `stack_switch()`, and `fpu_taskswitch()`. These are often bundled into a list of calls and passed as an argument to a single hypercall, `multicall()` for the performance optimization. Thus, measuring the duration of `multicall()` captures the longest hypercall execution time. The result of measurement is shown in Figure 5.9. There were six VMs with one 1ms slot assigned for each. VMs were launched and performed multiple intensive tasks to incur many context switches inside each VM. The average duration was 2.38 μs and the maximum was 34.62 μs. During the non-steady state phase (OS loading & initialization), the maximum duration of a hypercall was 104.1 μs.

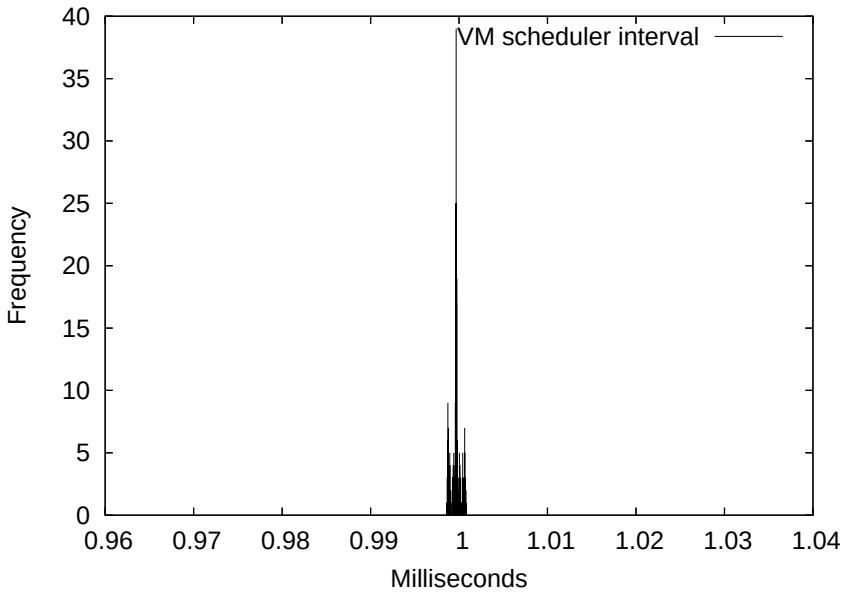

Figure 5.8: Periodic timer accuracy: VM-scheduling period (avg: 1.000 ms, max: 1.001 ms, min: 0.999 ms)

Other RT-VMM blocking times

Next, we measured the overhead of the periodic tasks of the RT-VMM. The major contributers to this number are the timer interrupt handler and the scheduler which run every 1 ms in our configuration. We scheduled all ten slots to a single VM. In this case, a run of scheduler does not entail VM context switching, so we can measure only the overhead incurred by the scheduler. We measured the performance of integer operations using a benchmark suit, `nbench 2.2.2`. The result was almost identical when we used the Linux's bogomips values. The result shows the average of 0.33 % performance loss over the native Linux environment. It translates to 3.33 μs of blocking time per 1 ms slot.

Another type of overhead of RT-VMM is VM context switching. We ran ten VMs with 1 ms slot assigned for each VM, and measured the aggregated performance by running *nbench* inside each and every VM for multiple

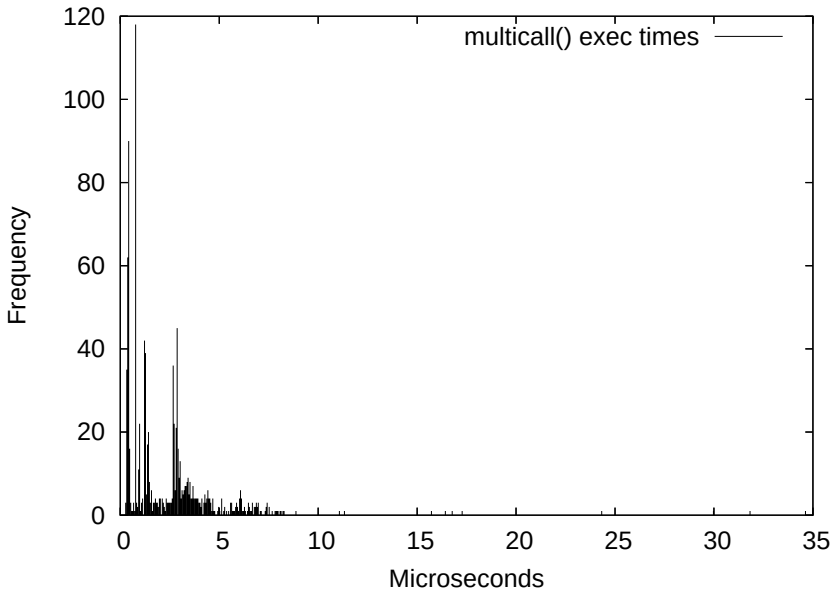

Figure 5.9: Hypercall execution times - multicall() (avg:2.38 μs, stdev 2.73, max 34.62 μs, min 0.28 μs)

times back-to-back in parallel. Since each VM runs only one user process, guest-level context switching was minimal. However, the measured overhead includes guest kernel's execution times, mainly the timer interrupt handling (10 ms period), and the scheduler, and a little bit of console I/O. The aggregated performance was 1.7 % slower than the native Linux environment, which translates to 16.93 μs of blocking time per 1 ms slot. This duration includes 3.33 μs of VMM's internal periodic executions described in the previous paragraph. The results are summarized in Table 5.1.

Network interrupts

Finally, we introduced the network interrupt. While the general server applications of VMM are mainly concerned with the throughput, we are more interested in how much overhead or blocking time a network interrupt generates. With our experimental setup of file transfer in an isolated switched network, the privileged VM with all-slot (10 out of 10) scheduled gets 1168 in-

	nbench int score	Overhead	Blocking times (μs/slot)
Native Linux	3.306	n/a	n/a
RT-VMM periodics	3.295	0.33 %	3.33
+ VM switchings	3.250	1.7 %	16.93
+ Net interrupts	3.150	4.7 %	47.19

Table 5.1: Blocking times. The rate of net interrupt is 1.13 ints/ms. i.e. (47.19-16.93)/1.13 = 26.78 μs/int

terrupts per seconds. (Between 3c905C-txm and eepro100, 1500 bytes MTU) When the performance of the single-slot (1 ms) scheduled VM is measured while the same transfer is on going, the privileged VM with nine-slot (9 ms) scheduled receives 1103 interrupts per second. The interrupt count is averaged over a 5 minute period. The performance of the single-slot scheduled VM is measured using nbench, which runs for a little over 10 minutes. The result shows that 30.26 μs is used for the interrupt processing out of its 1 ms slot time. Since there are 1.103 interrupts per 1 ms, the overhead is 26.78 μs per interrupt. The file transfer test would also incur disk controller interrupt, but we eliminated this by using a ram disk based file system. The OS-level ISR is run only when the privileged VM is scheduled. At a higher transfer rate, this can become a significant overhead. In order to keep the blocking time under a certain level, the rate has to be limited.

In summary, the typical blocking time experienced by the application process is 16.93 μs per 1 ms slot in our experiment. If network traffic is present, one interrupt causes additional 26.78 μs of blocking time. If the rate of network is limited, the total blocking times can be kept reasonably small.

6 VEER: Recovery using RT-VMM

VEER (Virtual Execution Environment for Robust Real-Time Systems) is a software architecture that utilizes RT-VMM for designing efficient recovery-enabled real-time embedded systems. The hardware-level virtualization allows execution of multiple operating system instances and guarantees each OS/VM meets all timing requirements. The RT-VMM prevents any resource contamination from one OS failure (i.e. crash or hang). Since embedded software often access special hardware, the potential failure can easily take the entire system down. With RT-VMM, such an incident can be isolated. In addition to the RT-VMM's protection, VEER provides a set of design guideline for organizing components and subsystems according to their criticality and dependency for better fault isolation and dependency management.

VEER is not for safety critical systems. Rather, it is more targeted to general real-time applications where use of COTS or open-source components without any quality guarantee is common. VEER's RT-VMM can host multiple instances of different operating systems, whereas RTVM architecture [81] is closer to the OS-level virtualization technologies such as OpenVZ [66] and Solaris 10 containers [33].

6.1 Differentiated recovery

The use of RT-VMM changes the traditional assumptions on the recovery. For normal systems, for instance, if the processing speed of a processor is 100

MIPS, the aggregated speed experienced by all the applications on the system cannot exceed 100 MIPS. Under a real-time virtual machine architecture, the top-level scheduler may allocate a "contingency slot" to prevent rare exceptions from causing deadline misses. This allows a virtual machine to run "faster" than the virtual speed limit to recover in time.

As illustrated in Chapter 2, the existence of explicit timing requirements on fault detection and recovery will certainly help improving the availability of real-time systems. However, setting aside a significant portion of CPU time for rare failure occasion is costly.

The use of RT-VMM allows to maintain a pool of slots for recovery that are shared among VMs. If no free slot is available, RT-VMM can suspend non-critical or non-real-time tasks to recover more critical real-time tasks. Who gets what depends on the system design and organization.

In the actual implementation, an additional hypercall is added to the existing RT-VMM in order to switch between normal mode and recovery mode. Two sets of schedule must be supplied to the scheduler by the designer for the two modes to work. An example is shown in Table 6.3. When the control VM detects or is notified of a failure-recovery (e.g. process resurrection of the inverted pendulum controller), it invokes `rtvm_sched_boost( )` hypercall to boost the corresponding VM according to the predefined schedule. The request for transitions from normal to recovery mode (i.e. boosting) can only be made in the control VM (i.e. VM0). Other VMs can make requests to end the local recovery mode. The control VM can end any VM's recovery mode. For Linux guests, the hypercall can be invoked via the proc file system interface, which is created by the driver, *rtvm_sched_boost*. This simplifies the interfacing user-mode monitoring daemons to VEER.

6.1.1 Failure detection

For a recovery to work, we must be able to detect failures as early as possible. There are different types of failures and each of them can be detected different ways.

When a generic central monitoring method is used, it may not be a best way to react to certain failure events. For example, if a periodic monitoring is used, the failure-detection latency can be as long as the period of the monitor. While periodic polling of system status works best for certain type of failures, this is not the case for failure modes such as crash. For crash failures detection and recovery, we use Process Resurrection, which is described in Chapter 3. Process Resurrection can be used for recovery only, if a predefined user signal is used as a custom trigger. Any other detection mechanism can trigger the fast recovery in this way. Also for real-time systems, non-crash failures are highly application specific and timing errors are often associated with the symptoms. We use *eSimplex* to mask the content (e.g. unacceptable results) and timing errors (e.g. deadline miss). If the overhead of running redundant controller is too high, an ORTGA-based [61] analytic redundancy may be possible for the application.

6.1.2 Criticality ordering

As described in Chapter 1, components must be partitioned according to their criticality levels so that non-critical components cannot make critical components fail. But if two components with different criticality levels communication with each other, how can we guarantee that the non-critical one is not going to adversely affect the critical one when a fault occurs?

The traditional dependency analysis can be used, but to guarantee a

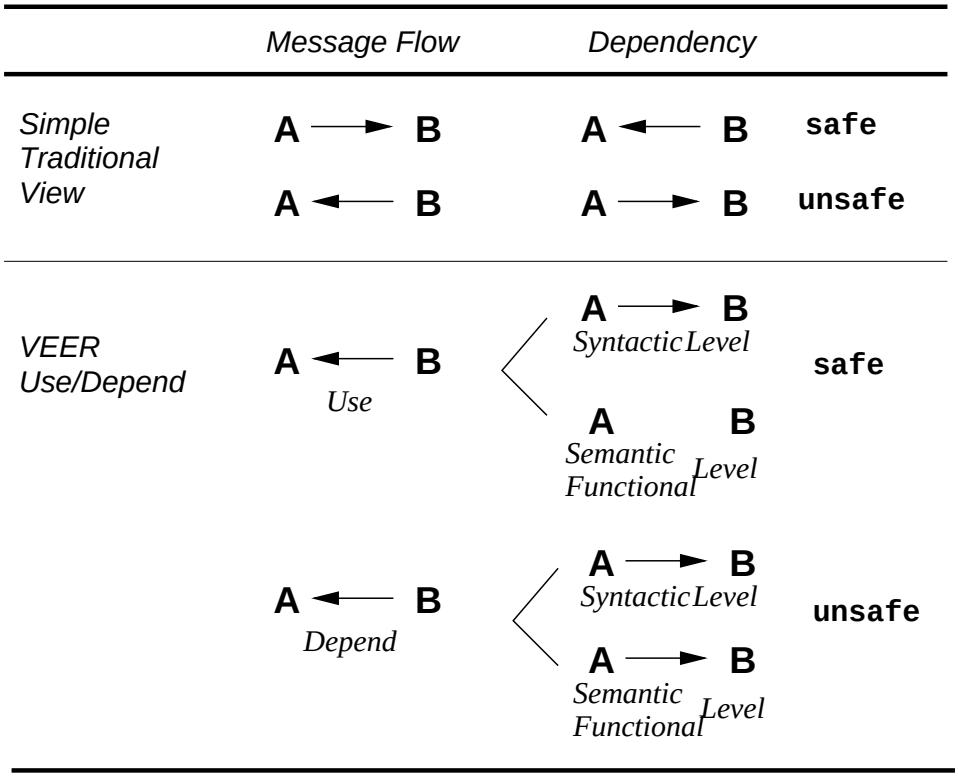

Figure 6.1: Traditional view of dependency Vs. Use/Depend concept

safety, it will impose too much restrictions on the system. For example, if component A is a critical component and component B is a non-critical one, the only way to ensure the safety is by showing A does not depend on B. In other words, there should be no connection from A's input ports to B's output ports, only allowing data flow from A to B and preventing ones from B to A. (See Figure 6.1)

This is a correct and sufficient condition that make sure our criticality ordering works, but can be too restrictive in some cases. For example, suppose that there is a critical component that checks outputs from one non-critical component against the one from a critical component, and chooses the output from non-critical components, only if the data is available and it passes the check, otherwise it chooses the one from the critical-component. In this

case, an incorrect or absent output from the non-critical component cannot adversely affect the critical monitoring component. But according to the traditional dependency graph, we are not allowed to design the system this way.

This restriction can be removed by looking at the semantic level dependencies as well as syntactic level dependencies. In above case, there is a syntactic level dependency, but no semantic level dependency. When a failure of a component has no effect on syntactically dependent components, we call it a *Use* relation. Thus if we can enforce a *Use* relation to non-critical to critical data flow, we know our criticality ordering rule is not violated and faults are isolated at lower criticality levels. A formal description of *Use* relation is available in [30].

In order to enforce a *Use* relation, application semantics must be taken into account. This is because a Use relation is determined by how an application is processing the incoming data. Thus it is impossible to have a generic *Use* enforcer that works for any application. We can, however, provide the hooks to static analysis tools such as [50] that perform architecture-implementation conformance tests. Such tests can also generate runtime checks for checking dynamic properties. Certain checks can be placed ouside the virtual machine to reduce the execution time variations cased by the checks.

It is also important to note that enforcing Use relations to data flows from non-critical to critical components is essential, having *Use* relations between components regardless of criticality levels also helps reducing recovery time or performance degradation by isolating faults and limiting the affected region in the system.

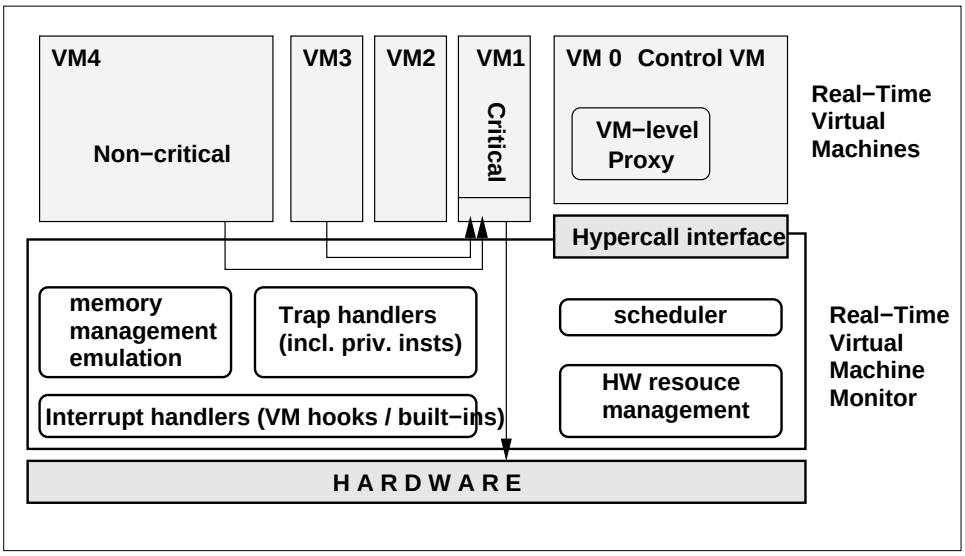

Figure 6.2: An example of VEER-based configuration

6.2 VEER Design Guidelines

Even though we can prioritize urgent recoveries over non-critical tasks or use free slots, it will take a much longer time to recover if faults have been propagated to other VMs and have corrupted them. In fact, many faults are not immediately detectable, so they might propagate widely until they are finally detected. Moreover, since faults can propagate through legitimate channels, architecture-level isolation alone cannot prevent the propagation. Fault occurs at different layers of abstraction. A comprehensive fault containment management requires the use of both VM level and application level fault containment measures. In VEER, we provide a set of guidelines to follow in order to better utilize the features of the RT-VMM architecture. An example will be provided in the experiment and evaluation section.

The key to contain a fault is the proper management of dependency [30]. Among many factors, we consider the following two: *criticality* and *capability*. (i) *Criticality:* It is important to make sure the critical components are

not depending on non-critical components. The first step is to categorize the system components into at least two criticality levels, critical and non-critical. For example, the computerized brake system in a vehicle can be regarded as a critical component, whereas the air conditioning system is important but non-critical. The next step is to make sure that the components with different criticality are not placed in the same container. (e.g. process, process group, or VM, depending on how the system is organized) For example, if a non-critical thread and critical thread belong to the same process, a segmentation fault by the non-critical component will also bring down the critical one. The final step is to follow the *Use-Depend* concept [80] to make sure the critical components do not depend on non-critical component. Offline analysis tools such as [50] can be used to enforce this in design time.

(ii) *Capability:* In normal operation modes, components with different capabilities can coexist without any problem. But when a failure occurs, more potent component can do much more damage. For example, X servers often access hardware and kernel memory directly for the improved performance and control. When it crashes and compromises the hardware or the kernel, other unrelated processes can also be affected. If we have the perfect failure models for every component in the system to predict their failure modes, we can use this information to separate the components. However, if such information is not available, we should consider the worst case by examining the capability of each component. The simplest separation will be between computation-only components and the ones with low-level hardware access.

The actual recovery method varies depending on the failure model of the components in a partition. For example, if operating system failures are taken into account, the recovery methods such as VM-restart and VM-hot-swap can be implemented using existing techniques. It is also important

Components	Criticality	Low-level	Real-time	Grouping
sensing actuation	high	yes	yes (10 ms)	VM1
HAC, Simplex	high	no	yes (20 ms)	VM2
HPC	low	no	yes (20ms)	VM3
UI	low	yes	no	VM4
video server	low	yes	soft	VM5
remote access	low	no	no	VM3

Table 6.1: The components in the system

note that not all components or VM require real-time recovery. The system requirement may allow delay of recovering certain real-time tasks, especially non-critical ones.

6.3 Experimental application

6.3.1 Organization

Our experimental system is composed of six subsystems or components as listed in Table 6.1. The first three are migrated from an existing system, and the others are either new or reimplementation of old components. We have classified each component according to its criticality and capability. First, the critical components are separated and placed in a separate VM. The first three components were originally running in the same process group on a single machine. In the old configuration, if the user-submitted HPC (high performance controller) embeds an assembly code that kills every process in the process group, the HAC (high assurance controller) and all others get killed. For this reason, we have enhanced the security with a compiler-based

technique. In the new configuration, a malicious HPC can only damage itself and the remote control server, which can be restarted in the background. In this way, fault containment has been improved.

Second, the ones with low-level hardware access capabilities are separated. If the UI component malfunctions, it can render the console unusable. For many systems, rebooting is the only solution. We are also rebooting the OS, but only for the specific VM. Although a reboot of the VM takes more than 10 seconds on our platform, other components are not at all affected by it.

Non-critical real-time and non-real-time components can coexist as long as the operating system can meet the timing requirement of the real-time component. In VM3, we have such a mixture. They are hard to separate because the remote access server launches user supplied HPCs.

The RT-VMM is depending on VM0 to carry out certain tasks. If VM0 crashes, the default behavior of RT-VMM is to reboot. However, this can be optionally disabled. In addition, it is recommended that the number of drivers in VM0 be minimized. If it can be avoided, no direct hardware driver should be there except perhaps the disk back-end drivers for launching other VMs. This is also unnecessary if RAM disk is used for the root file system. We use RAM disk-based root file systems for most critical VMs in the experiment.

A schedule is determined and built into the RT-VMM in build time. The schedule also includes the prioritized restart order and other details. The RT-VMM first uses empty slots for recovery, then non-real-time components' slots, and finally non-critical real-time components' slots. Every VEER-based system has VM0, a control VM, for bootstrapping and monitoring. It can be used for monitoring the health of other VMs. We have total of six of VMs in the system. The schedule and recovery plan are shown in Table 6.2

VM	Name	Normal	Boost	Critical	Real-time
0	super	3,7	1-10	Yes	No
1	phys-io	1	1	Yes	Yes
2	simplex	2	2,6,7	Yes	Yes
3	hpc	4	4,5,7	No	Yes
4	ui	5,6	5,6,7	No	No
5	vid	8,9,10	6-10	No	Yes

Table 6.2: The schedule and properties of each VM

and the actual configuration is shown in Table 6.3. VM0's recovery mode is enabled when the system first boots in order to get the system ready as soon as possible. Once VM0 is ready, it must be transition to normal mode and launch other VMs.

6.3.2 Recovery

Although we do not guarantee the recovery of critical components, the RT-VMM can try to recover them in time by suspending all others. In this example, however, a failure of VM1 will likely affect the hardware and the external physical device (inverted pendulum), and any attempt to recover only the software will be futile. The sensing and actuation component in VM1 can recover from a simple process crash using PR (Process Resurrection) [52]. However, if the hardware state is compromised by the failure, the only cure may be a reboot. Since a reboot of VM1 takes 10.17 seconds on average at the full CPU speed and the deadline is 10 ms, this cannot be a viable recovery method. VM-hot-swap may be possible, but software-only swap may not be able to recover the hardware, if the hardware cannot be safely and completely reset by software. In this case, software-originated critical hardware-related failures, which may require hardware redundancy for masking contaminated hardware.

```
#define SLOTS_PER_CYCLE 10
#define NUM_VM 6

short int sched_norm[SLOTS_PER_CYCLE] =
   {1, 2, 0, 3, 4, 4, 0, 5, 5, 5};

short int sched_recovery[NUM_VM][SLOTS_PER_CYCLE] =
{ {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},   /* VM0 */
  {1, 1, 0, 1, 1, 1, 0, 1, 1, 1},   /* VM1 */
  {1, 2, 0, 3, 4, 2, 2, 5, 5, 5},   /* VM2 */
  {1, 2, 0, 3, 3, 4, 6, 5, 5, 5},   /* VM3 */
  {1, 2, 0, 3, 4, 4, 4, 5, 5, 5},   /* VM4 */
  {1, 2, 0, 3, 4, 4, 5, 5, 5, 5} }; /* VM5 */

short int criticality_level[NUM_VM] =
{1, 1, 1, 2, 2, 2};

/* 0: can boost any. can unboost any
 * 1: can boost self. can unboost self
 * 2: can boost non. can unboost self
 */
short int boost_priv[NUM_VM] =
{0, 1, 1, 2, 2, 2};
```

Table 6.3: The actual schedule configuration of the experiment

The simplex/HAC module in VM2 is auto-restart enabled using PR. The average restart time is 614.1 μs and the worst case is 729.1 μs (Figure 6.3). It includes clean-up operations before restart and the initialization activities such as communication channel creation, setting up scheduler and signal handlers, and creating and programming an interval timer. If PR is not used, the fresh start-up takes 2.25 ms on average (Figure 6.4). The measured WCET of simplex/HAC in its steady sate is 63.38 μs (Figure 6.5). Even if we overestimate the WCET as 80 μs, the execution-recovery-re-execution cycle will take 889.1 μs, which is smaller than the usable slot size of 952.81 μs (1 ms - 47.19 μs blocking time per slot). In this case, we do not need to assign additional slots for recovery. Since the coverage of PR is limited to crash failures, the types of faults that do not cause crash are translated to "deadline misses" and detected externally by the helper VM. If the monitoring daemon does not hear from the VM2 components, it initiates a restart of the processes. In this case, we need to launch the processes from scratch and two more slots are needed before the end of the major cycle. This is the only VM where real-time recovery is needed.

The video encoding and streaming server is composed of two software components from an open source media processing software, FFmpeg: ffmpeg and ffserver. They are soft-real-time processes meaning that their deadlines are not strict, but if the deadlines are constantly missed, the latency of the streaming video will increase and eventually lose its usefulness. Its availability is not strictly required, but improves user experience. From the recovery point of view, there are two kinds of failure modes in this subsystem, the failure modes that require a process restart and the ones require a VM restart. Since the frame grabber hardware is being accessed by the software components in this VM, a software failure may corrupt the hardware state.

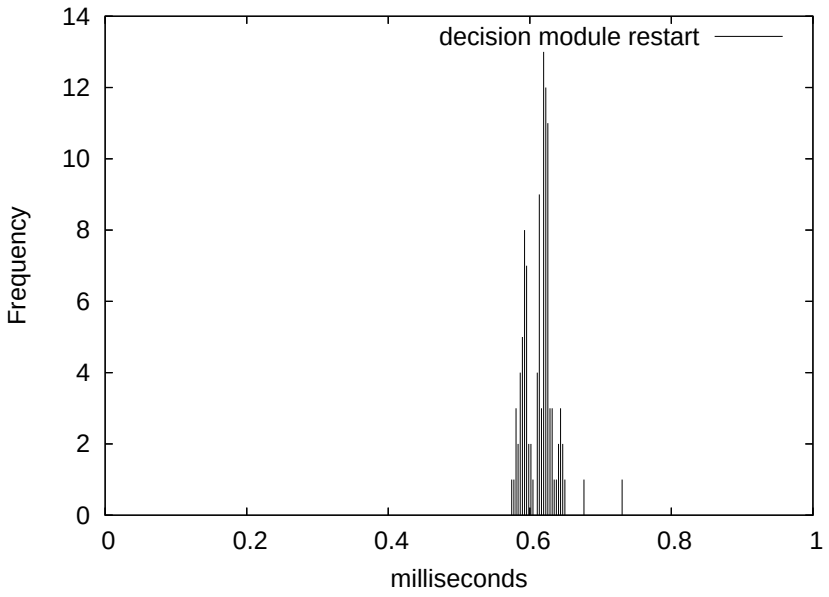

Figure 6.3: Restart times of the decision module using process resurrection

Some hardware problems can be solved by driver reloading, but not all OS-hardware combination support it. To simplify the recovery, we always reset the VM. It is also monitored by the helper VM and if VM5 crashes, two more slots are allocated to speed up the recovery. The reboot recovery with five slots takes about 18 seconds, and 30 seconds without the extra slots. The boot process is faster than the others because VM5 uses a root file system on a physical partition, not a compressed ram disk image.

The UI in VM4 is used by the local user for editing and submitting a custom HPC and viewing the video stream. The main component in this VM is the X server. The X server bypasses the video BIOS and directly manipulates the video card. It also accesses the kernel memory space. In most cases, a crash only needs restart of the X server, but there are cases that need restart of OS. The OS restart takes a little more than 10 seconds and launching X server/twm takes about 4 seconds at the full CPU speed. When rebooting, one more slot is assigned. The recovery time in this configuration

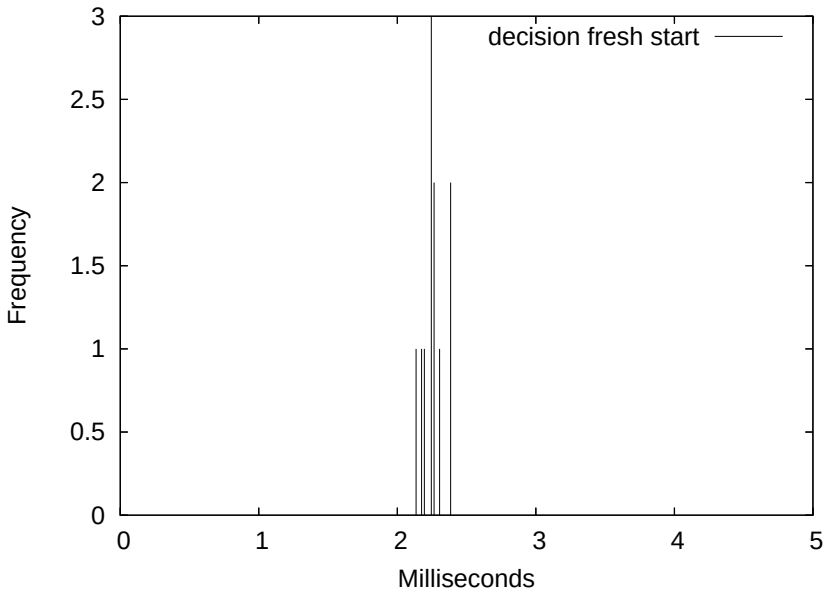

Figure 6.4: Fresh start times of the decision module

is 46 seconds. The user can also trigger a restart.

The components in VM3 are treated differently in recovery. Since the HPC component is submitted by an external user, it may contain malicious code. The examples of malicious controller code are taken from [59]. When the system is compromised and the service is not available (periodic polling of the service is performed by VM0), VM3 is rebooted. If the remote control server is killed, it tries to restart by itself through PR. If the system is not corrupt, the recovery will succeed without reboot. The HPC is submitted by users and is not part of the file system image. A user must submit a new HPC after the recovery if needed. A crash of HPC does not trigger any recovery action. Three more slots are allocated for recovery and it takes 37 seconds to reboot and restore the service. Without the slot boosting, it will take more than two minutes.

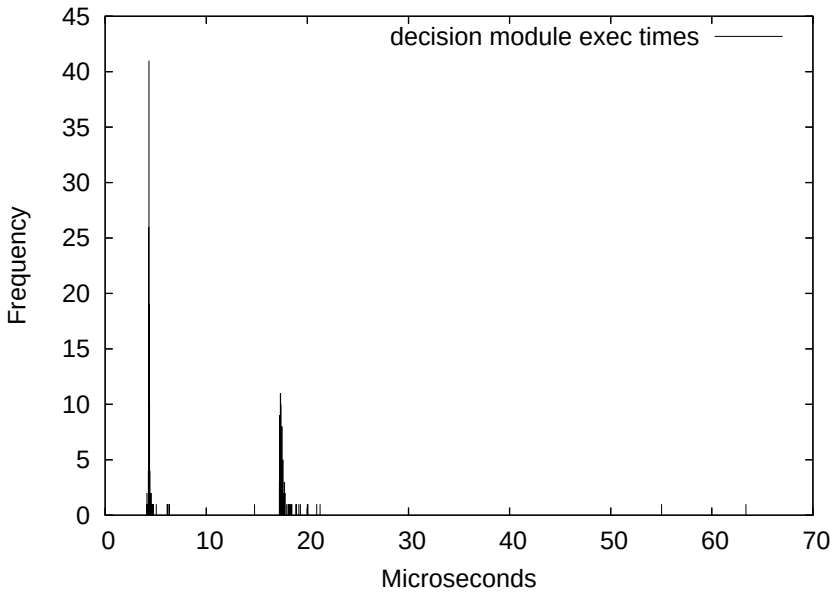

Figure 6.5: Execution times of the decision module (for WCET estimation)

6.3.3 Comparison

The migrated system was composed of multiple machines. The components in VM1, VM2, and VM3 were on one machine, and the VM4 and VM5 were separately running on two machines. In order for the comparison to accurately show how much better the recovery performance of VEER is, the aggregated resource of the original design must be comparable to the resource available on the machine running the RT-VMM. In our experiment, it was hard to meet this condition due to the unavailability of hardware that meets all the requirements at the comparable speed. The aggregated CPU clock speeds of the original system is 1266 MHz (400+133+733), while our new machine has a 750 MHz processor. The differences in the organization also make it difficult to directly compare certain qualities. For this reason, we do not directly compare numbers to numbers. Rather, we describe how they are different and the implications of the differences.

VM	#slots, normal	#slots, recover	Method	VEER recovery time	recovery time on equivalent machine
0	2	10	boot	n/a	n/a
1	1	n/a	n/a	n/a	n/a
2	1	3	restart	729us/2.25ms	729us/20.25ms
3	1	3	reboot	37sec	2min
4	2	3	reboot	46sec	70sec
5	3	5	reboot	18sec	30sec

Table 6.4: Comaprison of recovery performance

The partitioning by RT-VMM gives nearly the same degree of isolation we had in the old systems. If the virtual CPU speed of each corresponding VM is equivalent to the one in the original design, the normal case and reboot performance will be roughly the same between old and new design. But the original system cannot take the resource from the remote machines to speed up the recovery of the failed machine. This is obviously a unique benefit of VEER. In addition, the reorganization based on our guideline improves the robustness of the system by preventing potentially malicious HPC component from harming others, and failures in the components with low-level access privileges from being propagated. If all the components were run on the same operating system on a single machine, it will be hard to enforce and control each process's CPU usage and timing in the way RT-VMM does without a special operating system. However, more serious problem is the lack of fault isolation. As illustrated earlier, a crash of X server may require a system reboot. It may also compromise the critical components by corrupting the kernel or hardware state.

7 Conclusions

VEER enables strict control of VM timings in both normal and recovery mode with a reasonably small and predictable overhead, thanks to the use of RT-VMM. The fault isolation and containment can be improved by separating the components according to their criticality and the capability. The fault propagation is blocked at the component level using analytic redundancy techniques such as *eSimplex* in addition to the partitioning provided by RT-VMM. The improved isolation reduces the scope of recovery. Therefore faster and timely recovery is possible.

By utilizing VEER's dynamic scheduling policy switching capability, the share of CPU time can be manipulated. This results in a better control over how VMs are to be treated in special conditions such as failure-recovery. When compared to the hypothetical equivalent computers, the components in VEER can recover significantly faster without affecting critical components, even when rebooting. This gain comes at the cost of reduced performance in other less critical or important tasks. It is only possible because all tasks are running on a single machine.

The current version of VEER has limitations. As in Xen 2.0.7, it does not support the full `ioperm` operations, and the I/O space partitioning is still voluntary; if the guest OS in a privileged VM does not comply, there is not much we can do. Although IO-MMU-based solution will be ideal for providing partitioning and virtualization, we can at least provide a strong isolation in I/O space by using predefined fixed I/O bitmaps and not allow-

ing non-critical VMs to change it. Since the embedded hardware is mostly static, this approach will not be much of a problem. If the use of IO-MMU such as intel VT-d technology (no spec as been released at the time of writing) becomes common in embedded real-time systems, the implementation of more complete partitioning and virtualization will be justified.

The Intel processors with the new virtualization support have been released but are not widely used yet. Some machines are equipped with one, but are not enabled due to lack of BIOS support. If such a feature becomes standard in embedded systems, and a better virtualization with less performance overhead becomes possible, VEER will most likely add an option for full virtualization for the better compatibility. VEER inherited its paravirtualization architecture from Xen mainly for its performance advantages on existing hardware platforms. It is still unclear whether Intel VT-x-based full virtualization will perform better than paravirtualization. It will certainly a big win over emulation or binary translation-based virtualization techniques.

RT-VMM can be utilized in many other ways. Some require full virtualization. To list a few:

- Resource throttling. Its complete control over resource allocation can be used for CPU throttling and power management when the guest OS is not capable of doing it or there is a need to enforce certain policies.

- RT support for any non-RT systems. Non-real-time operating systems such as Microsoft Windows XP can coexist with a real-time operating system on one machine. It does not involve modifying non-real-time OS HAL or any other part, yet real-time processing can be carried out and the results can be transferred to the non-real-time OS. Single processor cell phone may one day run one real-time OS and another

non-real-time OS.

- Predictable prioritization. In consolidated non-real-time environment, RT-VMM can quantifiably and predictably prioritize certain VMs for recovery or other special conditions.

- Non-invasive dynamic reconfiguration. For real-time or multimedia systems such as home media hub, additional services can be safely plugged in or removed without affecting the performance of the existing ones.

It is this author's belief that introduction of real-time concepts to the non-real-time world opens a new opportunity to enable more fine grained controls. RT-VMM is certainly a promising candidate.

References

[1] V. Abrossimov, F. Hemann, J-C Hugly, F Ruget, E. Pouyoul and M. Tombroffet, *Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology,* CSI-T4-96-34 Technical Report, Chorus Systems, Inc., Aug. 1996.

[2] A. Agbaria, R. Friedman, "Virtual machine based heterogeneous checkpointing," *Proceedings of the International Parallel and Distributed Processing Sumposium*, 2002

[3] B. D. Aleksa, and J. P. Carter, "Boeing 777 airplane information management system operational experience, " *Proceedings of the 16th AIAA/IEEE Digital Avionics Systems Conference*, 1997.

[4] L. Alvisi, T. Bressoud, A. El-Khasab, K. Marzullo, and D. Zagorodnov, "Wrapping Server-Side to Mask Connection Failures," *Proceedings of INFOCOMM 2001*, vol 1, pp. 329-337, April 2001.

[5] T. Anderson, J. C. Knight, "A framework for software fault tolerance in real-time systems," *IEEE Transactions on Software Engineering*, SE-9(3):355-364, May 1983.

[6] A. Avizienis, "The Methodology of N-Version Programming," *Software Fault Tolerance*, M.R. Lyu, ed., John Wiley & Sons, New York, 1995.

[7] A. Avizienis, J.-C. Laprie, and B. Randell, "Dependability And Its Threats: A Taxonomy," *Building the Information Society*, pp.91-120, Kluwer Academic Publishers, 2004.

[8] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Communications of the ACM*, 36(11):81-94, Nov. 1993.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization", *SOSP 2003*, Oct. 2003.

[10] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," *Proceedings of 2005 Usenix Annual Techinical Conference*, April 2005.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the SPIN operating system", *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 267-283, Copper Mountain, Colorado, 1995.

[12] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan, *Linear Matrix Inequalities in system and Control*, SIAM, 1994.

[13] T. C. Bressoud, and F. B. Schneider, "Hypervisor-based fault tolerance," *SOSP 1995*, pp. 1-11, 1995.

[14] A. Brown, G. Kar, and A. Keller, "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment," in *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, USA, May 2001.

[15] A. Brown, and D. A. Patterson, "Embracing failure: A case for recovery-oriented computing (ROC)," *2001 High Performance Transaction Processing Symposium*, Asilomar, CA, Oct. 2001.

[16] G. Bylinsky, "Industrial Management and Technology: Heroes of Manufacturing," *Fortune*, 147(6), March 2003.

[17] M. Caccamo, G. Buttazzo, L. Sha, "Elastic feedback control," *IEEE Proc. 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, pp. 121-128, 2000.

[18] G. Candea and A. Fox, "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel," *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.

[19] G. Candea, A. Brown, A. Fox, and D. Patterson, "Recovery Oriented Computing: Building Multi-Tier Dependability", *IEEE Computer*, 37(11):60-67, November 2004.

[20] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - A Technique for Cheap Recovery", *6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[21] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan and W. P. Zeggert, "Proactive Management of Software Aging," *IBM Journal of Research and Development*, 45(2), March 2001.

[22] S. Chandra, P. M. Chen, "Whither Generic Recovery From Application Faults? A Fault Study using Open-Source Software," *Proceedings of DSN 2000*, June 2000.

[23] F. Chen and G. Rosu, "Java-MOP: A Monitoring Oriented Programming Environment for Java," *TACAS 2005*.

[24] L. Chung and N. Subramanian, "Architecture-based semantic evolution: a study of remotely controlled embedded systems," *Journal of Software Maintenance: Research and Practice*, 15(3):145-190, May 2003.

[25] *Computer,* 38(5), May 2005.

[26] J. E. Cook and J. A. Dage, "Highly reliable upgrading of components," *Proceedings of the 21st International Conference on Software Engineering*, pp. 203-212, 1999.

[27] A. Cunei, and J. Vitek, "A new approach to real-time checkpointing, " *VEE 2006*, June 2006.

[28] *Design guide for integrated modular avionics*, ARNIC Report 651-1, Aeronautical Radio Inc., Annapolis, MD, 1999.

[29] H. Ding, K. Lee and L. Sha, "Dependency Algebra: A Theoretical Framework for Dependency Management in Real-Time Control Systems," ECBS 2005.

[30] H. Ding, and L. Sha, "Dependency algebra: A tool for designing robust real-time systems," *RTSS 2005*, pp. 210-220, 2005.

[31] B.L. Di Vito, "A model of cooperative noninterference for integrated modular avionic", Proceedings of the 7th International IFIP Conference on Dependable Computing for Critical Applications, pp. 269-286, October 1999.

[32] B. Dutertre, and V. Stavridou, "A model of noninterference for integrating mixed-criticality software components", Proceedings of the 7th International IFIP Conference on Dependable Computing for Critical Applications, pp. 287-300, October 1999.

[33] P. B. Galvin, "Solaris 10 containers," *;login: The Usenix Magazine*, 30(5):11-14, Oct. 2005.

[34] S. Graham, G. Baliga and P. R. Kumar, "Issues in the convergence of control with communication and computing: Proliferation, architecture, design, services, and middleware," To appear *Proceedings of the 43rd IEEE Conference on Decision and Control*, Bahamas, Dec. 14-17, 2004. March 7, 2004.

[35] B. Hailpern, and P. Santhanam, "Software debugging, testing, and verification", *IBM Systems Journal*, 41(1): 4-12, 2002.

[36] M. Hicks, J. T. Moors and S. Nettles, "Dynamic software updating," *Proceedings of PLDI 2001*, Snowbird, Utah, June 2001.

[37] D. Hildebrand, "An architectural overview of QNX," *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, WA, April 1992.

[38] G. L. Holzman, "The power of 10: Rules for developing safety-critical code," *IEEE Computer*, 39(6): 95 - 97, June 2006.

[39] W. C. Hsieh, M. E. Fiuczynski, C Garrett, S. Savage, D. Becker, and B. N. Bershad, "Language support for extensible operating systems," *Proceesings of the First Workshop on Compiler Support for System Software*, February 1996.

[40] Y. Huang and C. M. R. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," *Proceedings of 23rd Intl. Symposium on Fault-Tolerant Computing*, pp. 2-9, June 1993.

[41] Y. Huang, C. Kintala, N. Kolettis and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proceedings of 25th Intl. Symposium on Fault Tolerant Computing*, pp. 381-90, 1995.

[42] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill, *An overview of the Singularity project,* Microsoft Research Technical Report MSR-TR-2005-135, Microsoft Corporation, Redmond, WA, Oct. 2005.

[43] *Intel Virtualization Technology for IA-32 Processors (VT-x) Preliminary Specification*, ftp://download.intel.com/technology/computing/vptech/C97063.pdf, Intel, April 1, 2005.

[44] D. Kim, Y.-H. Lee and M. Younis, "SPIRIT-μ-kernel for strongly partitioned real-time systems," *RTCSA 2000.*

[45] K. H. Kim, H. O Welch, "Distributed execution of recovery blocks: an approach for uniformtreatment of hardware and software faults in real-time applications," *IEEE Transactions on Computers*, 38(5):626-636, May 1989.

[46] M. Kim, I. Lee, U. Sammapun, J. Shin and O. Sokolsky, "Monitoring, checking, and steering of real-time systems," *2nd International Workshop on Run-time Verification*, Copenhagen, Denmark, July 26, 2002.

[47] R. Kirner, P. Puschner, and I. Wenzel, "Measurement-Based Worst-Case Execution Time Analysis," *Proceedings of 4th Euromicro International Workshop on WCET Analysis*, June, 2004.

[48] P. Koopman, and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems", *IEEE Transactions on Software Engineering*, 26(9): 837-848, September, 2000.

[49] S. Kowshik, D. Dhurjati and V. Adve, "Ensuring code safety without runtime checks for real-time control systems," *CASES 2002*, Grenoble, France, 2002.

[50] S. Kowshik, G. Rosu, and L. Sha, "Static analysis to enforce safe value flow in embedded control systems," *DSN 2006*.

[51] J. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*, 2nd ed., CMP books, June 2002.

[52] K. Lee, and L. Sha, "Process resurrection: A fast recovery mechanism for real-time embedded systems", *RTAS 2005,* pp. 292-301, Mar. 2005.

[53] K. Lee and L. Sha, "A Dependable Online Testing and Upgrade Architecturefor Real-Time Embedded Systems", *The 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

[54] K. Lee, *memo from the interview with Professor Sylvian Ray*, March 6, 2006.

[55] J. P. Lehoczky, Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *IEEE Real-Time Systems Symposium*, pp.201-213. 1990.

[56] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," *OSDI 2004*, pp.17-30. 2004.

[57] B. Lewis, E. Colbert and S. Vestal, "Developing evolvable, embedded, time-critical systems with MetaH," *Proceedings of TOOLS 2000*, 2000.

[58] F. Liberato, R. Melhem, and D. Mossé, "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-time Systems," *IEEE Trans. on Computers*, 49(9):906-914, Sep. 2000.

[59] S.-S. Lim, K. Lee and L. Sha, "Ensuring integrity and serice availability in a web-based control laboratory," *Parallel and Distributed Computing Practices*, 4(2):165-178, June 2001.

[60] J. W. S. Liu, K. J. Lin, and S. Natarajan, "Scheduling real-time, periodic jobs using imprecise results," *RTSS 1987*, Dec. 1987.

[61] X. Liu, H. Ding, K. Lee, L. Sha, and M. Caccamo, "Feedback Based Real-Time Fault Tolerance - Issues and Possible Solutions," *ACM SIGBED REVIEW*, 3(2), April 2006.

[62] J. Luke, J. W. Bittorie, W. J. Cannon, and D. G. haldeman, "Replacing Strategy for Aging Avionics Computers", *IEEE AES Systems Magazine*, pp. 7-11, March, 1999.

[63] LynxOS-178, http://www.lynuxworks.com/rtos/rtos-178.php, LynuxWorks, Inc.

[64] T. J. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric," *NIST Special Publication 500-235*, Computer Systems Laboratory, MD, September 1996.

[65] J. E. O'Neill, "Prestige luster and snow-balling effects: IBM's development of computer time-sharing," *IEEE Annals of the History of Computing*, 17(2):50-54, Summer 1995.

[66] OpenVZ, http://openvz.org.

[67] P. Oreizy, N. Medvidovic and R. N. Taylor, "Architecture-based runtime software evolution," *Proceesings of the ICSE 1998*, pp. 177-186, Kyoto, Japan, April 1998.

[68] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*, UC Berkeley Computer Science Technical Report UCB-CSD-02-1175, March 15, 2002.

[69] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures", Communications of the ACM, 17(1):412-421, July 1974.

[70] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac, and A. Wellings, "GUARDS: Generic Upgradable Architecture for Real-Time Dependable Systems", *IEEE Transactions on Parallel and Distrubuted Systems*, 10(6): 580-599, June 1999.

[71] Presentations and discussions at the *2nd LMCO-UIUC Workshop on System Integration*, Urbana, IL, July 29-30, 2004.

[72] H. Myrén, J Piculell and L. Lundberg, "Run-time upgradable software in a large real-time telecommunication system," *Proceedings of the 7th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2001.

[73] T. J. Ostrand, and E. J. Weyuker, "The distirubtion of faults in a large industrial software system," *ISSTA 2002*, pp. 55-64, July 22-24, 2002.

[74] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, SE-1:1220-232, 1975.

[75] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor", *Proceedings of 9th USENIX Security Symposium,* pp. 129-144, Aug. 2000.

[76] J. Rushby. *Partitioning for safety and security: Requirements, mechanisms, and assurance,* NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.

[77] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, 39(9):1175-1185, Sept. 1990.

[78] L. Sha, R. Rajkumar, and S.S. Sathaye, "Generalized rate-monotonic scheduling theory: A framework for developing real-time systems," *Proceedings of IEEE*, 82(1), Jan. 1994.

[79] L. Sha, R. Rajkumar, M. Gagliardi, "Evolving dependable real-time systems," In Proc. of *1996 Aerospace Applications Conference*, pp. 335-346, Feb 1996.

[80] L. Sha, "Using simplicity to control complexity," *IEEE Software Magazine*, 18(4):20-28, July 2001.

[81] L. Sha, "Real-time virtual machines for avionics software porting and development," *RTCSA 2003*, pp123-135. 2003.

[82] J. A. Stankovic, "Strategic directions in real-time and embedded systems," *ACM Computing Surveys*, 28(4): 751-763, December 1996.

[83] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for nextgeneration systems," *IEEE Computer*, 21(10):10-19, Oct. 1988.

[84] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMWare workstation 's hosted virtual machine monitor," *Proceedings of 2001 Usenix Annual Techinical Conference*, June 2001.

[85] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, 39(5):44-51, May 2006.

[86] K.S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova, "Modeling and analysis of software aging and rejuvenation," *Proc. 33rd Annual Simulation Symposium*, pp. 270-279, Apr. 2000.

[87] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the Denali isolation kernel," *Proceedings of USENIX OSDI 2002*, December 2002.

[88] N. Woo, "Multimedia embedded systems for mobile applications", Keynote presentation at *11th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, March 8, 2005.

[89] A. P. Wood, "Software reliability from the customer vie", *IEEE Computer*, 36(8):37-42, Aug. 2003.

[90] K. M. Zuberi, P. Pillai and K. G. Shin, "EMERALDS: a small-memory real-time microkernel", Proceedings of the seventeenth ACM symposium on Operating systems principles, pp. 277-299, 1999.

Author's Biography

Kihwal Lee was born in September 26, 1971 in Seoul, Korea. As early as 1980, he started assembling electronic circuits by obtaining parts from then flourishing Cheong-gye-cheon parts shops. The elementary school yearbook lists his prospective profession as an "electrical engineer with a Ph.D."

His interests were broaden after encountering the world of ham radio in 1985. He started building RF circuits, mostly shortwave receivers and accessories. In 1988, he published his first technical article on BFO construction for SSB reception on *KARL Magazine*, the monthly publication of the Korean Amateur Radio League. His long wish to learn computers became reality when his parents purchased an IBM XT clone while he was in high school. In 1990, he entered the Department of Environmental Engineering (then Department of Occupational Safety and Health) of Yonsei University. While he loved the subject and excelled, it could not subdue his passion for computers and electronics.

He stayed and finished the sophomore year after returning from the 28 months of mandatory military service. Shortly after, he transferred to Southern Illinois University at Carbondale, where he earned his B.S. in computer science in 1998. He continued his graduate study at the University of Illinois at Urbana-Champaign under the supervision of Professor Lui Sha. Kihwal Lee's interests include real-time embedded systems, robust software, and history of computing. He has been involved in the development of projects such as eSimplex, Telelab, wireless sensor-based acoustic tracking, Process Resurrection, and real-time virtual machine monitors. He completed his Ph.D in 2006.